Software Engineering for Parallel and Distributed Systems

Edited by

Innes Jelly, Ian Gorton and Peter Croll

IFIP

SPRINGER INTERNATIONAL PUBLISHING, CHAM

Software Engineering for Parallel and Distributed Systems

IFIP – The International Federation for Information Processing

IFIP was founded in 1960 under the auspices of UNESCO, following the First World Computer Congress held in Paris the previous year. An umbrella organization for societies working in information processing, IFIP's aim is two-fold: to support information processing within its member countries and to encourage technology transfer to developing nations. As its mission statement clearly states,

> IFIP's mission is to be the leading, truly international, apolitical organization which encourages and assists in the development, exploitation and application of information technology for the benefit of all people.

IFIP is a non-profitmaking organization, run almost solely by 2500 volunteers. It operates through a number of technical committees, which organize events and publications. IFIP's events range from an international congress to local seminars, but the most important are:

- the IFIP World Computer Congress, held every second year;
- open conferences;
- working conferences.

The flagship event is the IFIP World Computer Congress, at which both invited and contributed papers are presented. Contributed papers are rigorously refereed and the rejection rate is high.

As with the Congress, participation in the open conferences is open to all and papers may be invited or submitted. Again, submitted papers are stringently refereed.

The working conferences are structured differently. They are usually run by a working group and attendance is small and by invitation only. Their purpose is to create an atmosphere conducive to innovation and development. Refereeing is less rigorous and papers are subjected to extensive group discussion.

Publications arising from IFIP events vary. The papers presented at the IFIP World Computer Congress and at open conferences are published as conference proceedings, while the results of the working conferences are often published as collections of selected and edited papers.

Any national society whose primary activity is in information may apply to become a full member of IFIP, although full membership is restricted to one society per country. Full members are entitled to vote at the annual General Assembly, National societies preferring a less committed involvement may apply for associate or corresponding membership. Associate members enjoy the same benefits as full members, but without voting rights. Corresponding members are not represented in IFIP bodies. Affiliated membership is open to non-national societies, and individual and honorary membership schemes are also offered.

Software Engineering for Parallel and Distributed Systems

Proceedings of the First IFIP TC10 International Workshop on Parallel and Distributed Software Engineering, March 1996

Edited by
Innes Jelly
Sheffield Hallam University
Sheffield
UK

Ian Gorton
CSIRO Division of Information Technology
Australia

and

Peter Croll
University of Sheffield
Sheffield
UK

SPRINGER INTERNATIONAL PUBLISHING, CHAM

First edition 1996

© 1996 IFIP International Federation for Information Processing

Originally published by Chapman & Hall in 1996
Softcover reprint of the hardcover 1st edition 1996
ISBN 978-1-5041-2948-0   ISBN 978-0-387-34984-8 (eBook)
DOI 10.1007/978-0-387-34984-8

A catalogue record for this book is available from the British Library

∞ Printed on permanent acid-free text paper, manufactured in accordance with ANSI/NISO Z39.48-1984 and ANSI/NISO Z39.48-1984 (Permanence of Paper).

CONTENTS

PREFACE

In March 1996, the First IFIP Workshop on Software Engineering for Parallel and Distributed Systems was held in Berlin, Germany. The workshop was co-sponsored by the German Computer Society GI (Gesellschaft fur Informatik), and organised in association with the International Software Engineering Conference (ICSE 18). This publication is based on the proceedings of the workshop.

The aim of the workshop was to provide a forum for exchange of information and publication of the latest technological and theoretical advances in software engineering for parallel and distributed systems. Our previous experience of running short workshops on this topic in Aachen, Germany (1993), Como, Italy (1994) and Hawaii, USA (1995) had indicated that there was a growing need for this specialised event. The International Programme Committee was formed from a group of experts in different countries and application areas, all of whom were enthusiastic to explore and publicise contemporary research in parallel and distributed software engineering.

Many software applications require the use of explicit parallel programming techniques in order to meet their specification. Parallelism is needed to exploit the processing power of multi-processor systems in order to achieve high performance, to provide fault-tolerance and reliability in safety-critical and real-time systems, and to deal with physically distributed computing resources. While the range of existing software and hardware technology that can be employed in parallel and distributed systems development is massive, a set of underlying problems concerned solely with the use of parallelism can be identified.

These include:
- identification of problem-domain and solution-domain parallelism
- incorporation of concurrent activities in specification and design
- architectural influences on design and implementation, including use of virtual machines
- correctness and testing of systems
- performance prediction, monitoring and evaluation of systems
- systems heterogeneity

Importantly, software engineers must deal with these issues in addition to tackling the more commonly identified problems which occur in all software projects. Within a number of different application areas, issues regarding the integration of good software engineering practice into the production process are increasing seen as highly relevant. These include science and engineering applications, real time control and distributed systems. Thus the emerging community of researchers involved in parallel software engineering is concerned with the impact that the specific requirement to handle concurrency has on the software development process. Topics for investigation include the requirements analysis and specification, design, implementation and verification of parallel and distributed software.

Over sixty papers were submitted to the workshop. Of these, twenty one were accepted as full technical papers, and presented at the workshop. All papers were fully reviewed, and we are very grateful to all the referees for their careful and knowledgeable evaluation of each paper.

Contributions based on the application of new techniques to real software systems were encouraged as well as those describing advances in theory or methods.

As well as technical paper presentations, two special review sessions were included in the workshop. The first of these involved demonstrations of a number of software support tools: these were documented as short papers and included within these proceedings. In order to encourage dissemination of information on current collaborative projects, five project review papers were provided.

The papers cover a wide range of topics, and represent some of the most recent work in the area of parallel and distributed software development. New methods, tools and theories are described and offer solutions for some of the problems identified above. Together they constitute a view on the-state-of-the-art in parallel and distributed software engineering. Several papers address high level design approaches for parallel and distributed software. Themes emerging from these include the necessity to model communications, the use of object based methods and the role of CASE technology. At the implementation stage, work on automatic parallelisation is reported in two papers, and issues of configuration and scheduling are also explored. The verification of parallel and distributed software forms the topic of three papers: these explore tool support for early verification and testing, and, more formally, the role of temporal logic in model checking. Within the formal modelling area, papers describe new theoretical work as well as the application of techniques to different systems, and the role of process algebras is clearly demonstrated in these. Performance prediction and analysis is of crucial concern within many parallel and distributed software projects. Four papers present new work on the classification of performance issues, performance modelling techniques and analysis of algorithms.

For the future, we believe that this book has highlighted a number of issues. There is a need for rigorous design methods which encompass the range of abstractions required to specify and implement concurrent software. The parallel and distributed software community has recognised that good tool support is an essential aspect of the engineering process but there has been insufficient emphasis on inter-operability between different tools. Better integration of tools and techniques would offer practitioners a more robust approach to software development. Performance is a key issue for the majority of parallel and distributed systems, and further work on temporal representations is thus required to support formal specification and analysis. Also, within the formal modelling area, greater emphasis should be given to increasing the accessibility of existing techniques to allow software developers to use these in a coherent manner. This will involve both the provision of automated support and better levels of abstraction in modelling methods.

We would like to thank all members of the International Programme Committee not only for their work in organising the review process but for all the helpful and good humoured contributions towards the setting up of the workshop. Thanks are also due to the organisers of ICSE-18 for their support for the workshop, including provision of administrative and registration facilities.

Innes Jelly
(Programme Chair)
i.jelly@shu.ac.uk

Ian Gorton
(Co-Chair)
iango@syd.dit.csiro.au

Peter Croll
(Co-Editor)
p.croll@dcs.shef.ac.uk

PROGRAMME COMMITTEE

Innes Jelly	(UK)	Chair
Ian Gorton	(Australia)	Co-chair
Arndt Bode	(Germany)	
Manfred Broy	(Germany)	
Helmar Burkhart	(Switzerland)	
Peter Croll	(UK)	
Ian Foster	(USA)	
Juergen Ebert	(Germany)	
Cherri Pancake	(USA)	
Brigitte Plateau	(France)	
John Potter	(Australia)	
Stefano Russo	(Italy)	
Naoshi Uchihira	(Japan)	

PART ONE

Research Papers

1

Infrastructural Software for Model Driven Distributed Manufacturing Systems

Ian Coutts, Marcos Aguiar and John Edwards
Loughborough University of Technology
Manufacturing Systems Integration Research Institute
Loughborough, Leicestershire LE11 3TU - England.
Tel.+44 1509 228250, Fax +44 1509 267725
Email I.A.Coutts@lut.ac.uk, WWW http://msiri.lut.ac.uk

Abstract

The modelling of manufacturing systems has become a necessary part of the drive for manufacturing enterprises to remain competitive within a global marketplace. Manufacturing models are increasingly being used not just as a means of articulating system design but also to drive manufacturing systems at run time. To achieve model execution within a distributed and integrated manufacturing system a number of infrastructural software elements are required.

1 INTRODUCTION

When preparing a paper for this workshop the authors attemped to contribute 'something from the real world'. The following section is a description of a typical industrial problem domain, for which researchers in the application of IT to manufacturing, are deriving solutions in the form of methodologies, software tools and supporting models. This description provides a context against which the CASE tool and infrastructural software described in the paper can be considered.

2 A TYPICAL INDUSTRIAL PROBLEM DOMAIN

In the electronics manufacturing industry the design of fine line printed circuit boards (PCB) implementing highspeed logic designs is a complex problem involving groups of personnel with a variety of skills (Berri, 1994). The process involves the generation of PCB layout designs, their simulation and analysis, followed by inevitable redesign, in an iterative cycle.

This environment of continual evolution requires the discipline of identifying and controlling design versions, in order to maintain integrity and traceability. Figure 1 describes a scenario

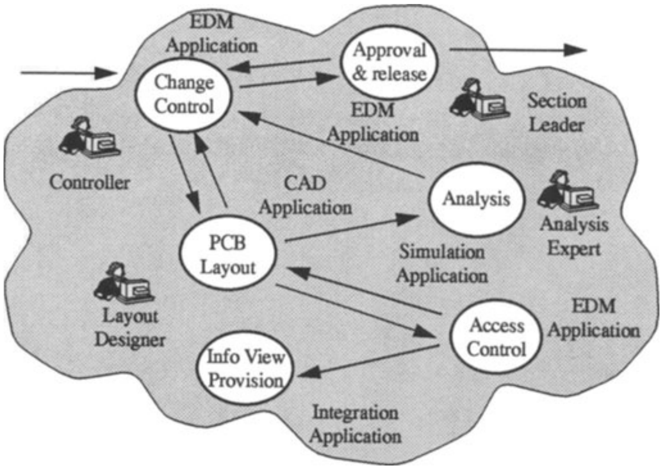


Figure 1. An Industrial Application Scenario

where personnel involved with engineering data management (EDM) must combine with those involved in the design and analysis processes using software applications from a range of vendors.

Required change to a PCB design could be initiated from an external request for a new variant of an existing product. A change control application driven by an engineering department operator could initiate a PCB layout change following completion of the electronic logic design. Following modification to the PCB layout, the company analysis expert will use a range of simulation applications to check for problems such as ringing, crosstalk and EMC (Berri, 1994). Throughout this process file access is controlled by another EDM application, while global information access is enabled through the use of an information view provision application (Weston, 1994). Following successful completion of the new PCB layout the design will be approved and released for prototype production by a senior member of the Engineering Department.

In order to provide a flexible integrated manufacturing software system to support this type of operation there is a requirement to model and rapid prototype a system based on a coordinated set of distributed software objects, where these objects interoperate through message passing. This formal approach to manufacturing system creation is part of a process often described as manufacturing enterprise engineering.

3 AN OBJECT-BASED BOTTOM UP MODELLING TOOL

The Bottom Up tool is intended to be used to model systems by combining manufacturing resources, while separating system behaviour from system function. Behaviour is captured using petri-net models and is executed at runtime, this behaviour is distributed as it is

described within the component objects of a system.
Figure 2 outlines the structure of the object-based CASE tool which was built using IPSYS

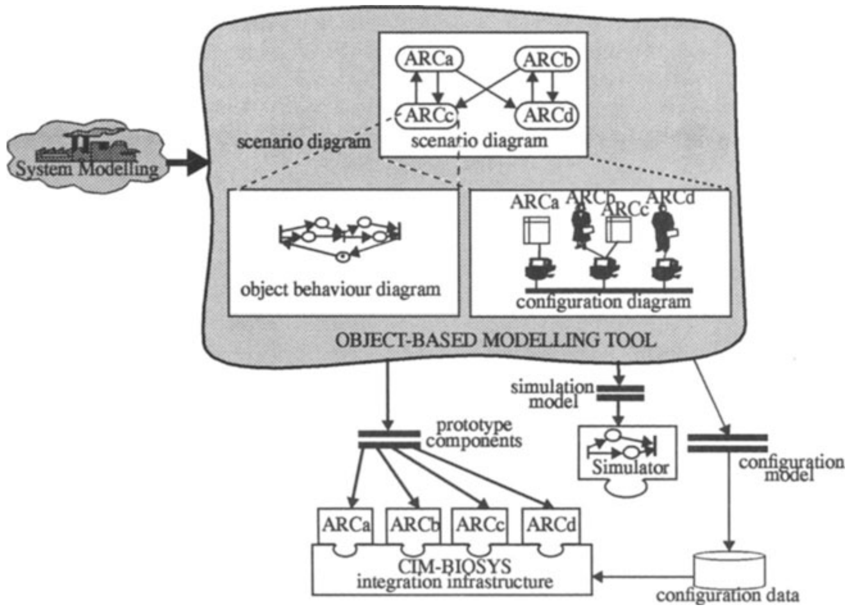


Figure 2. Object-based bottom up modelling tool

Meta CASE technology (Alderson 1991), the figure also shows the main elements of code that the tool can generate. Two aspects are of particular importance in this figure, namely: the design process embodied in the diagrams that the CASE tool supports, and; a capability for generating a rapid-prototype of a particular design which can be executed upon a layer of software which forms an integration infrastructure.

The modelling method encapsulated by the CASE tool comprises three diagramming techniques namely: a scenario diagram (which identifies the major objects (i.e. Active Resource Components* - ARC) which constitute the main components of a system being modelled, and defines the flow of messages among these objects), an object behaviour diagram (which defines the expected external behaviour of an object as perceived by the objects with which it interacts. Such a description utilises a predicate-action Petri-net (David, 1994) to define the internal sequence of actions within the object, as well as the relationships between such actions and external interactions with other objects to which the object in question relates) and a Configuration diagram. (which defines the computer configuration of the system

*. An active resource component identifies a component of a system which is able to execute an element of functionality on its own. It can also be a modelling description which characterises either a human being, an application program or a machine that possess a computerised controller. In the case of this paper, particular interest is placed upon active resource components which characterise software objects. The term Active Resource Component is defined within the CIM-OSA Reference Architecture (ESPRIT/AMICE, 1993)

i.e. on which host computer each active resource component will be executed).

These three diagrams constitute a minimal modelling facility for describing a system comprised of a number of objects which interact with one another in order to perform a common task. Such a description focuses on capturing the behavioural aspects associated with the way in which interactions occur within the system. The actual functionality performed by each object is defined in close association with the predicate-action Petri-net which defines its internal behaviour. This functionality comprises a set of object methods or object member functions which within this paper will be referred to as methods.

4 MODEL CREATION

As stated earlier, in order to engineer an agile* and integrated, distributed manufacturing software system there is a requirement to model and rapid prototype a system based on a coordinated set of distributed software objects, where these objects interoperate through message passing. The methodology embodied in the bottom up CASE tool can provide support for this process. The level of support can be illustrated by examining a general scenario. Figure 3 shows a representation of such a system, modelled via an object-oriented

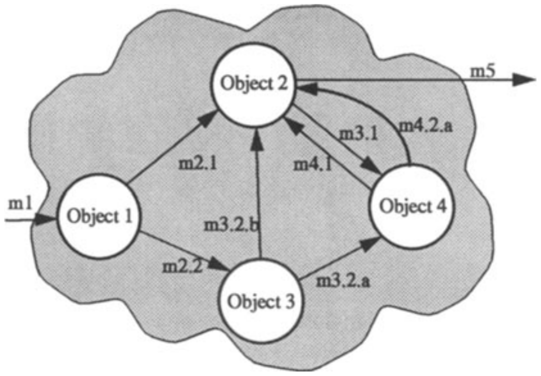


Figure 3. Scenario diagram of a hypothetical system

representation, where four objects are required to coordinate in order to support a distributed manufacturing process.

The objects depicted in Figure 3 perform tasks defined by their internal functionality where these are triggered by messages. This is described by the behaviour diagrams populated with the predicate-action Petri-net models as shown in Figure 4, this functionality may, of course, involve the actions of a human operator using the software object. Figure 4 also illustrates a proposed notation that explicitly classifies the type of action associated with the firing of each transition.

*. An agile manufacturing system embodies a high degree of long term flexibility. This provides support for system update and change which enables the manufacturing enterprise to respond quickly to changing market situations.

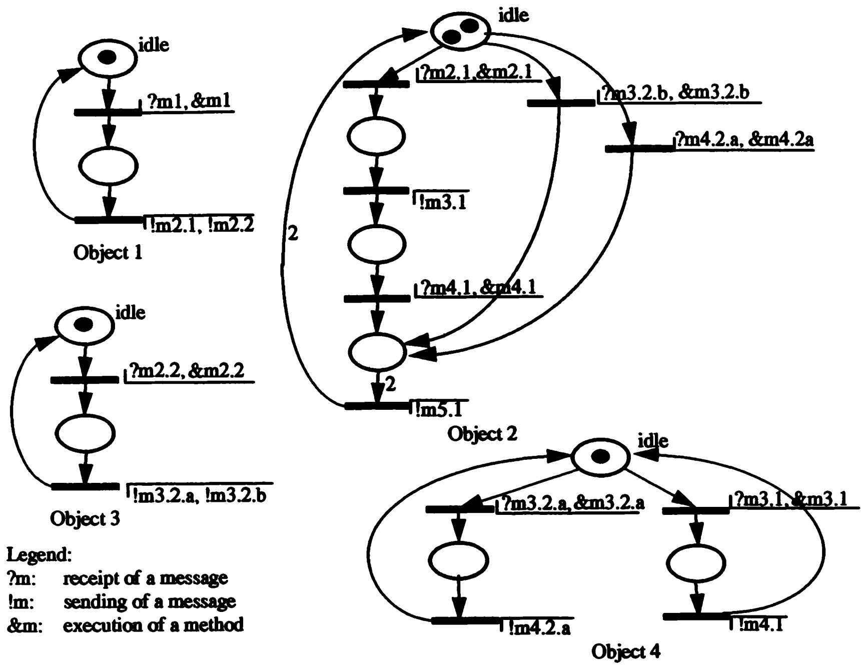


Figure 4. Petri-Net descriptions of the components of the original system

5 CODE GENERATION FOR RAPID-PROTOTYPING

Design information formalised through models produced using the CASE tool are passed in the form of a number of pieces of interpreted code to the infrastructural software that enables model enactment.

This paper now focuses on describing the infrastructural software elements which enable the models produced during system design to be executed during rapid prototyping.

6 MODEL EXECUTION

To facilitate the rapid prototyping of system solutions generated by the CASE tool. researchers at the MSI Research Institute have produced an environment which comprises a set of infrastructural software elements that sit above generally available operating system software. The principal components forming this environment are shown in Figure 5. The figure shows a number of objects interacting via an integration infrastructure, these objects are executable representations of the objects modelled within the scenario diagram of the CASE tool. Each object comprises four distinct areas of functionality as shown in the 'exploded' object in Figure 5, and as described in the following points:

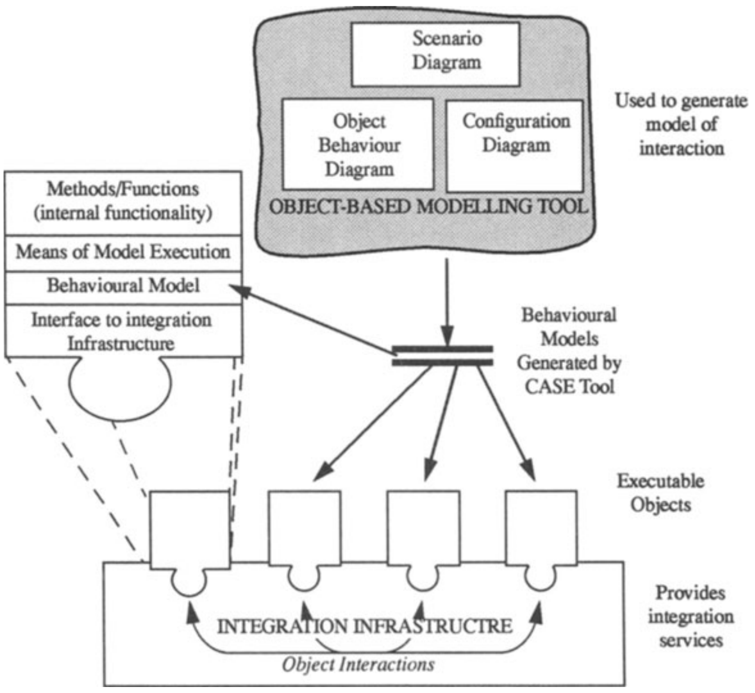

Scenario Diagram
Object Behaviour Diagram
Configuration Diagram
OBJECT-BASED MODELLING TOOL
Used to generate model of interaction
Methods/Functions (internal functionality)
Means of Model Execution
Behavioural Model
Interface to integration Infrastructure
Behavioural Models Generated by CASE Tool
Executable Objects
Provides integration services
INTEGRATION INFRASTRUCTRE
Object Interactions

Figure 5. Composition of Executable Objects

- an integration infrastructure interface which enables the object to access the integration services offered by an integration infrastructure.
- a behavioural model which describes the interactions between any one object and all other objects in the system. This is defined within the Object Behaviour Diagram of the CASE tool during the system design phase as a predicate-action Petri-net.
- a number of methods which comprise the internal functionality of an object.
- the means to execute both the object's behavioural model and trigger its internal methods.

The following sections focus on the structure of the objects identified in Figure 5 and the infrastructural software elements used to execute such objects. In order to describe these objects it is first necessary to briefly describe the integration infrastructure on which they interact (a detailed treatment of which is given in (Coutts, 1992)) and to describe the format of the behavioural models which are automatically generated by the CASE tool.

7 THE INTEGRATION INFRASTRUCTURE

An integration infrastructure provides the necessary services to enable interaction between the various software objects which comprise a system. It provides a consistent set of services irrespective of the objects's physical location, or the operating system and network protocols used. Researchers at MSI have created such an integration infrastructure called CIM-BIOSYS (CIM Building Integrated Open SYStems). The objects achieve interaction by using the integration services offered by the infrastructure. As well as object interaction (message passing), CIM-BIOSYS provides services for file access, data access and system configuration. Services pertinent to this paper include; EST_APP which establishes a peer to peer link between two software applications, TERM_APP which terminates a peer to peer link, STAT_APP, which obtains the status of peer to peer links, SEND_APP which sends untyped data to a connected peer.

The system configuration is produced by the CASE tool via the configuration diagram (see Figure 2). This details configuration information such as which objects reside on which platform, which executable image to invoke for a particular object and which network driver to use to communicate with a particular host. This information is held in a number of ASCII files (produced directly by the CASE tool) which are loaded when the infrastructure is initialised.

8 THE FORM OF THE OBJECT BEHAVIOURAL MODELS

The link between system design (or model building), and model execution is provided by the production of an executable version of the behavioural model created for each object described using the object behaviour diagrams within the CASE tool (as shown in Figure 5). Object behavioural models are represented by predicate-action Petri-nets during the system design phase within the CASE tool and are converted into a textual language (defined by researchers at MSI) during rapid prototyping. It is this textual language which can be enacted to facilitate model execution.

The textual language known as BTL (Behavioural Transition Language) defines predicates to describe interaction with other objects and predicates to execute internal methods. An example textual model for the predicate-action Petri-net Object2 defined in Figure 4 is shown in Figure 6. The language also provides for the representation of global variables which are

```
variable(idle,2).
transition( init1 , ( idle > 0 & recv_object(object1,"m2.1") ) ,
                                    ( method(2.1) @ p1 is p1 + 1 @ idle is idle - 1 ) ).
transition( init2 , ( idle > 0 & recv_object(object3,"m3.2b") ) ,
                                    ( method(3.2) @ p3 is p3 + 1 @ idle is idle - 1 ) ).
transition( init3 , ( idle > 0 & recv_object(object4,"m4.2a") ) ,
                                    ( method(4.2) @ p3 is p3 + 1 @ idle is idle - 1 ) ).
transition( p1-2, ( p1 >= 1) , ( send_object(object4,"m3.1") @ p2 is p2 + 1 @ p1 is p1 - 1 ) ).
transition( p2-3, ( p2 >= 1& recv_object(object4,"m4.1") ) ,
                                    ( method(4.1) @ p3 is p3 + 1 @ p2 is p2 - 1 ) ).
transition( p3-idle, (p3 >= 2 ), ( send_object(external,"m5.1") @ p3 is p3 - 2 @ idle is idle + 2 ) ).
```

Figure 6. BTL Model for Petri-Net description of Object 2

used for token counts, arithmetic expressions etc. The following provides some examples of such predicates:

method(2.1) - this executes the method called 'method 2.1'.
send_object(external,"m5.1") - this sends the message "m5.1" to the object called 'external'.
recv_object(object4,"m4.2a") - this tests to see if message 'm4.2a' has been received from the object called 'object4'.

p2 <= 1 - this tests to see if the variable p2 is less than or equal to 1.
p3 is "hello" - this sets the value of p3 to 'hello'.

A behavioural model expressed using BTL consists of a list of transition descriptions comprising a label, a condition and an action, which take the following form:

```
transition(label, condition, action).
```

The label is used to identify a particular transition, but serves no function within a BTL model. If the condition is satisfied the associated action is fired, composite conditions using the operator '&' to denote the AND operation can be used and a list of separate actions can be fired. Every action within the list is executed irrespective of the success or failure of the previous action, here the operator '@' is used to denote this 'follow on' operation. A list of variable initialisations can also be included. If variables are not initialised they are instantiated when they are first encountered and given a value of zero. As an example the following transition

```
transition(init2, (idle > 0 & recv_object(object4,"m4.2a")),
                                        (method(3.2) @ p3 is p3 + 1 @ idle is idle - 1)).
```

is labelled 'init2', and will execute method '3.2', add one to variable 'p3' and subtract one from variable 'idle', if 'idle' is greater than zero and the message 'm4.2a' has been received from object 'object4'.

9 THE STRUCTURE OF THE EXECUTABLE OBJECTS

Figure 7 provides a more detailed breakdown of the functionality within the four principal elements of an executable object. The figure also shows the important relationships between these functional elements. The principal contribution made while researching the requirements for the infrastructural software elements needed to support the bottom up model driven approach are highlighted in the two shaded areas on Figure 7: namely the 'model execution engine' constructed using Prolog* (Clocksin, 1984) and the 'event driven integration infrastructure interface' constructed using 'C' (Kernighan 1978). These two areas of functionality are described in the following sections, but for completeness and to establish the role of each component, the sections below describe each of the four areas from top to bottom as shown in Figure 7.

9.1 The methods which implement the internal functionality of the object.

As introduced earlier in the paper these are well defined and bounded pieces of functionality which perform the operations required for the object to fulfil its purpose. Such methods are identified by the modelling process and the 'trigger points' for their execution are contained within the object behavioural model generated by the CASE tool. As shown in Figure 7the methods are invoked by requests made by the execution engine. The code required to perform these methods is not automatically generated by the CASE tool and must be included by the system implementor. Within the current implementation such functionality can be included as Prolog source code or compiled 'C' object modules. Using the example outlined at the

*. The Prolog selected to implement the executable object was 'C Prolog' (as defined by Fernando Pereira, July 1982, EdCAAD, Dept. of Architecture, University of Edinburgh) this was chosen as it allows extension to both its Prolog environment and the 'C' source code in which it is written.

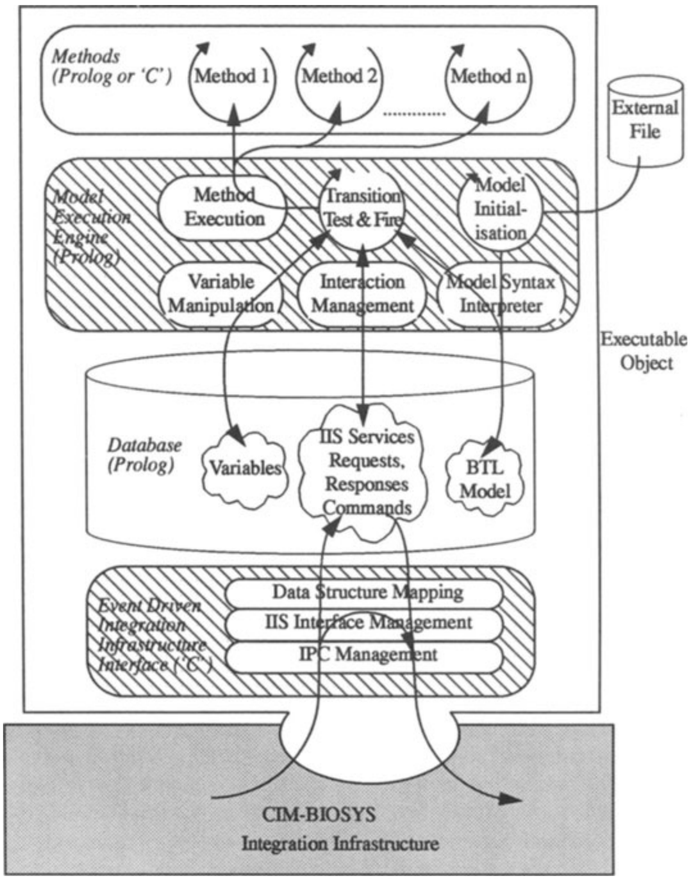


Figure 7. Structure of Executable Object

beginning of this paper, if an object constituted the change control part of a EDM system a internal object method might time-stamp a particular design.

9.2 A behavioural model execution engine.

Executable objects are invoked via the CIM-BIOSYS EST_APP integration service, the name of the file containing the BTL version of the appropriate object behavioural model is supplied at this time. The model import and initialisation process then loads the BTL model from the host computer's file system into the object's internal database. If required peer connections are then established and the transition test and fire process is started. This process continually tests the conditional part of all the transitions contained within the BTL model. If the result of a condition evaluation is true the corresponding action or actions are fired. The subsequent execution of predicates contained within the transition descriptions causes the executable object to either: manipulate internal variables e.g token counts within the behavioural model, execute internal functionality via the predefined methods or interact with its external environment through message passing via the use of integration services provided by the integration infrastructure (CIM-BIOSYS).

As shown in Figure 7 the model execution engine contains the two processes (model initialisation and transition test and fire) explained above and the following four separate functional support modules.

Model syntax interpreter
This provides both access and parsing of the BTL model held within the Prolog database. The transition test and firing process uses this functional support to examine the transitions held within the BTL model. This functionality has been mainly achieved via extensions to the resident Prolog parser.

Interaction management
This provides access to the integration service requests, responses and commands held within the Prolog database. The transition test and firing process can use this functional support module to issue integration service requests to the database, these will then be sent to the infrastructure by the integration infrastructure interface. The opposite is true for incoming messages, they appear via the integration services from the integration infrastructure into the database and then onto the execution engine.

Variable manipulation
This provides access to and manipulation of the global variables defined within BTL. These variables are manipulated by operators defined by the Prolog language. However the scope of the variables does differ from those defined by the Prolog language, as once they are initialised they can be shared and manipulated by all predicates and transitions until they are explicitly removed i.e they are global in nature.

Method execution
This enables the transition test and firing process to invoke the predefined internal object methods associated with the underlying function of the object.

9.3 A Database.

This serves the executable object as a data repository which holds: a) the BTL model loaded from the external file system; b) any associated variables required to execute the BTL model and; c) integration service requests or responses generated by the model execution engine or the integration infrastructure interface. The database also serves as an interface between the model execution engine and the integration infrastructure interface as both components are driven by the integration service requests held within the database.

9.4 An Integration Infrastructure Interface.

This is an event driven interface which provides the functionality required to allow the executable object to access the integration services offered by CIM-BIOSYS. This interface also makes the executable object appear as a CIM-BIOSYS compliant application by communicating with CIM-BIOSYS using the required protocol and inter-process communication mechanisms. The interface has been constructed to respond to two different types of event. One event type is the arrival of a data packet from CIM-BIOSYS, typically this could be an incoming integration service command or a response to a service command initiated by the executable object. In both cases the corresponding request is constructed and instantiated in the database. Another example is the arrival of a low level "heartbeat" to check the object is still operating as expected in which case an appropriate reply is issued. The other event type is the instantiation of an integration service request within the database in this case the corresponding integration request is constructed and delivered to CIM-BIOSYS and the request removed from the database.

As shown in Figure 7 the integration infrastructure interface contains the following three separate functional support modules.

Data Structure Mapping

This maps between the form of the structured data packets required by CIM-BIOSYS integration services and that defined in BTL. It also provides the mechanism by which asynchronous access to the database is enabled. This insures incoming integration service requests are processed concurrently with the transition testing and firing process.

IIS Interface Management

This manages the necessary interaction with CIM-BIOSYS through achieving the following: executing object initialisation sequences; by performing any handshaking required during normal operation; by executing the object termination sequence; by the handling of status requests from other applications, and; by the routing of integration service requests between the internal database and CIM-BIOSYS.

IPC Management

This is a facility which uses UNIX Inter-Process Communications to facilitate the exchange of structured data packets between the executable object and CIM-BIOSYS.

The executable objects described within this section provide a means for the execution of behavioural models generated by the CASE tool. Additionally they provide an environment in which a number of implementation features of the whole model execution process can be tested and enhanced. The model execution engine can be readily modified to accommodate changes in behavioural model syntax, behavioural model functionality and integration services used. This flexibility is principally due to the engines construction being based on the Prolog programming language. However, flexibility does not compromise runtime performance or ability to function in a distributed environment due to the event driven integration infrastructure interface being implemented using the 'C' programming language.

10 CONCLUSIONS

This paper has introduced a CASE tool for the design, implementation and execution of integrated manufacturing systems. The paper has focused on a description of the infrastructural support software which provides enactment facilities for the behavioural models created by the CASE tool. The distributed systems created using the tool and its infrastructural support elements benefit from a model driven approach which provides support for the key manufacturing enterprise need of accurate implementation of user needs and an improved ability to respond to required change.

The CASE tool and the infrastructural software described provide the support which brings together conventionally separate life cycle phases, from 'detailed design' to the 'configuration and execution' of systems upon an industrially tested integration infrastructure.

11 REFERENCES

Alderson A. (1991) Meta-CASE Technology. Lecture Notes in Comp. Science Software Dev. Env. and CASE Technology. Proc. of Euro. Symp. p81-91. Springer-Verlag. Germany.

Berri, J. (1994) High Speed Heaven, Printed Circuit Design.

Clocksin W.F. and Mellish C.S. (1984) Programming in Prolog, 2ed., Springer-Verlag.

Coutts, I. A. et al. (1992). Open Applications within Soft Integrated Manufacturing Systems, Proc. of Int. Conf. on Manufacturing Automation, Hong Kong, ICMA 92.

David, R. And Alla, H. (1994) Petri-nets for modelling of dynamic systems - a survey. Automatica, vol. 30, no. 2, pp. 175-202.

ESPRIT/AMICE. (1993) CIM-OSA Architecture Description, AD 1.0. 2.

Kernighan B.W, Ritchie D.M. (1978) The C programming Language, Prentice-Hall.

Weston RH, Clements P, Murgatroyd IS. (1994) Information modelling methods and tools for manufacturing systems, Proc. Lean/Agile Manufacturing in the Automotive Industries Conf. of the 27th Int. Symposium on Advanced Transportation Applications (ISATA), Aachen, Germany, pp227-234, ISBN o 947719 70 9.

12 BIOGRAPHY

I A Coutts spent two years at Marconi Research as a research scientist, working on industrial assembly automation and robotics projects. He has spent the last seven years as a member of the Loughborough University Systems Integration Group, and currently works in the MSI Research Institute at Loughborough. Particular responsibilities include work on infrastructural software and facilities for enabling model enactment.

M W Aguiar spent seven years as managing director in charge of the Integrated Automation Division of a Research and Development Institute of the Federal University of Santa Caterina/ Brazil. He has spent the last three years as a member of the MSI Research Institute, involved in the 'Model-Driven CIM' project, with particular responsibility for the conception, realisation, application and evaluation of SEW-OSA.

J M Edwards gained his Ph.D from Loughborough University in 1994. Having spent 13 years in UK process and manufacturing industry, being involved in the creation of computer control and information systems, he joined Loughborough University in 1987. During his 8 years at Loughborough he has been involved with the Systems Integration Group and is now a member of the MSI Research Institute where his role is as principal investigator on a number of UK government funded research initiatives.

2

Methodology and Tools for the Development of High Performance Parallel Systems with SDL/MSCs

Andreas Mitschele-Thiel
Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Informatik VII, Martensstraße 3, 91058 Erlangen, Germany,
email: `mitsch@informatik.uni-erlangen.de`

Abstract

The *Specification and Description Language* SDL and *Message Sequence Charts* (MSCs) are widely used in the telecommunication industry to support the software development process. In the paper, a methodology and a set of tools are described for the development of high performance parallel systems in the context of SDL and MSCs. While SDL and MSCs only support the formal specification of functional aspects of the system, we propose (1) the extension of MSCs to include non-functional requirements as the performance requirements of the application and (2) the annotation of SDL specifications with the respective execution cost on the parallel system. The formalization of non-functional aspects yields a set of benefits for system development: it allows the full integration of performance issues in all phases of the design process, starting from the requirements specification down to the final parallel implementation. It supports the automatization of performance related design decisions and allows the use of sophisticated tools supporting the performance optimization process.

Keywords

Parallel systems, software engineering, SDL, MSC, performance optimization, tools

1 INTRODUCTION

Parallel and distributed systems are inherently more complex than sequential systems. This is mainly due to the asynchronous execution of interrelated activities on different hardware units. In addition, the lack of a central control makes programming, debugging and testing of such systems extremely cumbersome. In order to lift these low-level activities to a more abstract phase of the software engineering process, the Specification and Description Language SDL has been introduced. SDL allows for taking corrective actions at a higher level of abstraction. This in turn reduces the cost of corrections by the order of magnitudes.

SDL has been standardized by ITU (ITU, 1993). In conjunction with tools, SDL is used by the majority of the companies in the telecommunication industry, mainly to design communication protocols and distributed applications. In addition, it is employed for the design of real-time and safety critical systems.

The latest version of SDL, SDL'92, with it's support for object orientation, supports the software engineering process from object-oriented design down to the generation of executable code. In conjunction with Message Sequence Charts (MSCs) (ITU, 1993b), system simulation and testing is supported, too. Besides a number of proprietary tools and tools from academia, there are two main providers of commercial tools for SDL, namely Telelogic with SDT (Telelogic, 1995) and Verilog with GEODE (Verilog, 1994). The tools support formal specification, validation, simulation, code generation and testing. While the tools for specification, validation, simulation and testing are widely used, the generation of the implementation is often done manually. This is due to the inefficiency of the code generated by the tools. In addition, implementations generated by the tools typically consume considerably more memory. In contrary, the manual implementation of SDL specifications contradicts the intended purpose of SDL and forces intensive testing of the application at the implementation level in order to ensure consistency with the specification.

A related problem is the lack of a formal approach in the system development cycle that supports non-functional requirements, e.g. performance or fault-tolerance requirements. This becomes even more obvious when an SDL specification is implemented on parallel systems due to the wide variety of design decisions that have to be met. These design decisions include (but are not limited to) the architecture of the parallel system, the distribution of code and data as well as the strategies employed for scheduling and dynamic load balancing.

In the paper a methodology is presented for the development of high-performance parallel systems with SDL and MSCs. The approach fully integrates performance issues in the system development cycle. The topic is highly relevant since it allows the fast development and modification of parallel systems in the scope of SDL which provide the required performance. Especially in telecommunications, a highly competitive market, the time to market has become the major issue to ensure competitiveness.

The paper is organized as follows. In section 2, an introduction to SDL and MSCs is given. In addition, the performance relevant issues with the engineering of parallel systems in the context of SDL are discussed. Our methodology for performance relevant development of parallel SDL systems is described in section 3. In section 4, the DO-IT toolbox supporting our methodology is described. Section 5 summarizes the paper.

2 ENGINEERING HIGH PERFORMANCE PARALLEL SYSTEMS WITH SDL AND MSC

2.1 Introduction to SDL and MSC

SDL specifications are fully hierarchically structured as a tree. The root of the tree refers to the SDL system specification which typically consists of a set of blocks. Blocks themselves can be refined by other blocks or by SDL processes. However, each leaf of the tree has to be an SDL process. The communication structure between SDL processes is static. Thus, all potential communication channels have to be given in the SDL specification.

SDL processes communicate asynchronously by exchanging signals. For each SDL process,

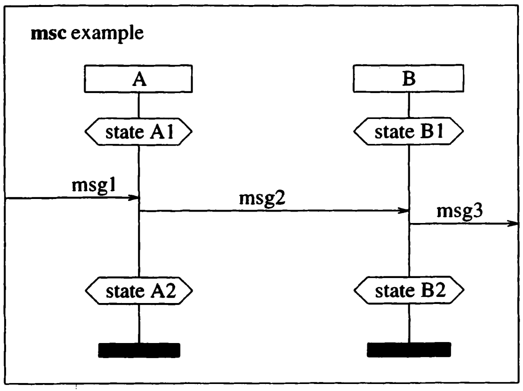


Figure 1 An example of an MSC

several process instances may exist which can either be static or created dynamically by other SDL processes during runtime. Each process instance owns a FIFO input queue.

Each SDL process represents an extended finite state machine (EFSM). For each state of the EFSM a set of trigger conditions is specified, typically the reception of a signal. If a trigger condition holds, a set of actions is performed, typically including the (asynchronous) sending of signals to other SDL processes. As a result of the actions, a subsequent state is entered.

SDL comes in two syntactic forms, the textual representation SDL/PR (SDL Phrase Representation) and the graphical representation SDL/GR. A detailed introduction to SDL can be found in (Braek and Haugen, 1993) and (Olsen et al, 1994).

In the development cycle, SDL is employed for the functional design of the system. The SDL specification focuses on the structural aspects of the application under development and the dynamic behavior of each of its processes.

In order to complement the SDL specification, MSCs have been proposed. MSCs represent a more abstract view on the system. An MSC describes the dynamic behavior of the system, i.e. one example (or instance) of a possible execution of the system. Thus, an MSC typically specifies how a message is passed through the entities of the SDL system, i.e. its blocks or processes. As a result, an MSC defines a partial order on the execution of the SDL system.

An example of an MSC, passing messages between the two processes A and B and the environment is given in figure 1. The MSC specifies the names of the messages passed and the states of the two processes before and after the transaction. MSCs are typically created during the requirements analysis, i.e. before the SDL specification. MSCs are mainly used to

- formally specify the functional requirements,
- serve as a basis for the generation of SDL skeletons and
- serve as a basis for testing.

A detailed introduction to MSCs can be found in (Olsen et al, 1994) and (ITU, 1993b).

2.2 Design Decisions in the Context of the Design Process

During the design phase, a high-level SDL specification is derived from the given MSCs. Then, the high-level SDL specification and the MSCs are subsequently refined to form the functional design specification. The check of consistency of the SDL specification with the MSCs is supported by tools. For the functional design also non-functional requirements need to be considered, e.g. to meet performance or safety requirements.

Designing parallel SDL systems, as well as parallel systems in general, numerous design decisions need to be made and appropriate alternatives chosen. In the following, we describe the decisions most relevant to the performance of the parallel SDL systems. We present the design decisions in the order in which they are made in the design process.

In the *SDL specification* the following design decisions are made

- the granularity of the SDL entities, i.e. what kind of functionality is provided by an SDL process, block or system,
- the parallelization technique, e.g. farming, functional decomposition or data decomposition,
- the dynamic of the system, i.e. the question whether SDL processes are created dynamically during runtime upon request or not,
- the representation and distribution of the data, i.e. whether data are maintained by a single central SDL process or distributed over the system and maintained by appropriate coherence protocols, and
- the use of (costly) SDL constructs.

The *code generation* is concerned with the physical (static) distribution of the software on the hardware. Thus, it determines which function is provided by which hardware unit. Since the function mapping may be one-to-many, the actual decision where to execute a specific function invocation resulting in a certain load on the hardware unit may be deferred to runtime.

The mapping of functions to hardware units intrinsicly decides the mapping of data. The reason for this is that in SDL, data are always encapsulated in processes. The code generation is also concerned with the granularity of the processes managed by the operating system. This has an important influence on performance especially if the cost of process management is high. Note that the code generation is only concerned with the merge of a set of SDL processes to form a single operating system process, and not the reverse problem, i.e. the parallelization of a single SDL process. Another decision that influences the code generation is the question whether the minimization of the execution time or the minimization of the required memory space of the application is the primary goal.

In addition, the code generation determines how the SDL constructs are mapped on relevant service primitives provided by the runtime environment (or the operating system in case no runtime environment is present). Most important is the implementation of communication primitives and primitives for the dynamic creation of SDL processes.

The *runtime library* provides the runtime environment for the SDL system, supporting the SDL constructs. The runtime library is typically provided in part by the SDL tools and has to be completed by the user. The part typically provided by the user comprises the implementation or mapping of the primitives for interprocessor communication and process creation on the respective primitives of the operating system. Here the most appropriate services with respect to functionality and performance are to be selected.

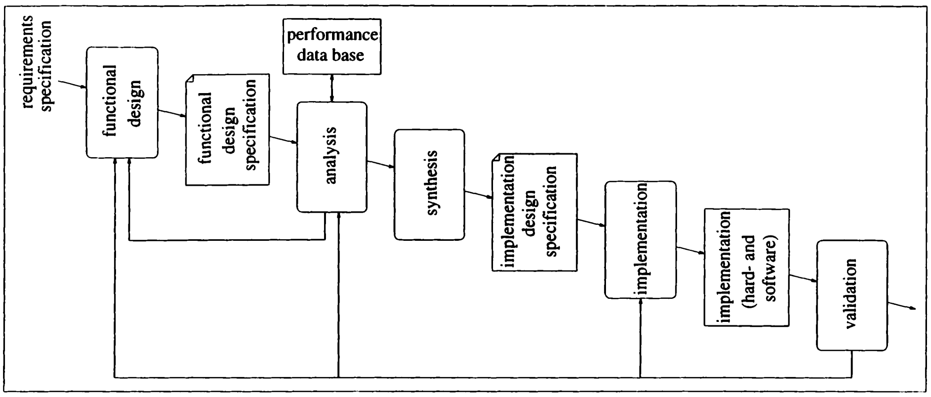

requirements specification
functional design
performance data base
functional design specification
analysis
synthesis
implementation design specification
implementation
implementation (hard- and software)
validation

Figure 2 The development methodology

In addition, strategies for

- the dynamic distribution of the load on the processors of the system and
- the dynamic scheduling of processes

either have to be implemented in the runtime library or at least the parameters to configure the respective services need to be provided. This is the case if the mechanisms for load balancing and scheduling as provided by the operating system are employed.

Decisions related to the *operating system* are the provided functionality, especially the support for parallel processing and communication, and its performance.

The *remaining code* provided by the user is mainly concerned with the implementation of abstract data types, the handling of exceptions and the handling of communication with the external interfaces.

3 METHODOLOGY FOR SYSTEM DEVELOPMENT

The major goal of our methodology is the early integration of performance data into the development process in order to minimize the time and cost for redesign and reimplementation. The outline of the methodology is depicted in figure 2. The part of the development cycle our methodology covers comprises five phases: functional design, analysis, synthesis, implementation and validation. Thus, our methodology covers the same part of the development cycle typically supported by SDL and MSCs.

Starting point of our methodology is the requirements specification. The requirements specification is subdivided in two parts, the functional and non-functional requirements. The functional requirements are formally specified with MSCs. The performance requirements of the system under development are also given formally. For this, an extended version of MSCs is employed. In addition, the machine architecture should be formally specified or the constraints on the machine architecture. The formal specification of the performance requirements and the machine architecture are a prerequisite for the automization of the design process.

Functional Design During the functional design phase, the functional design specification is derived. The functional design is specified in SDL and represents a functional description of the system at a detailed level. It is typically derived in a series of steps subsequently moving from a top- to a low-level design document. In conjunction with the refinement of the SDL specification, the MSCs, as given in the requirements specification, are subsequently refined to reflect the internal behavior of the refined SDL specification. The functional design specification should be checked for consistency with the requirements specification and for completeness before moving to the next phase. For the functional design, first tools are available to derive SDL skeletons from MSCs. In addition, the validation of the consistency with the requirements specification and the check for reachability and deadlocks is supported by commercial tools.

The functional design specification typically moves from a rather implementation-independent specification to a more implementation-specific specification. However, the initial functional design specification may already contain solution-domain parallelism. This is especially true when the need for parallelization is obvious from the requirements specification. For example, if a centralized data base is obviously not able to satisfy the required performance, it does not make sense to specify a centralized data base that will later on be replaced by a distributed implementation. The main reason for the straight development of the distributed data base is that a central data base, which will definitely be simpler, can in general not be used as a reference model to verify the distributed data base. This is because of the state space explosion problems typically encountered with the formal verification of extended finite state machines. Thus, the distributed data base is typically validated with the MSCs given in the requirements specification or a refinement hereof. This eliminates the need and usefulness of the centralized data base specification.

Analysis The purpose of the analysis phase is to check whether the functional design specification is sufficient to meet the given performance requirements, and in case this does not hold to provide guidelines to the user for a redesign of the functional design specification. The main performance criteria to be analyzed are the throughput and the response-time measures of the SDL specification. For this, the execution of the SDL specification on the machine(s) as specified in the requirements specification, with the load specified by the extended MSCs, is assumed. In order to support this, we pursue the extension of the SDL specification with annotations that allow for the specification of the execution time of SDL constructs on a set of available machines. In other words, each SDL construct is attributed with a vector. Each element of the vector specifies the execution time of the construct on a specific machine. Note that the term "machine" refers to an abstract machine, comprising the processor hardware, the runtime environment and the operating system. Thus, the execution cost given for an SDL construct is influenced by these components as well as the code generator. In case different code generators or operating systems are at the disposal of the design process, a separate machine model is employed for each of the combinations.

In order to derive the performance data, two approaches are possible. The cost of the execution of the SDL specification, or more specifically the constructs of the SDL specification, may either be derived analytically or by means of measurements.

For the analytic approach, a performance data base is needed that specifies for each SDL construct the respective cost to execute the construct on each of the available machines. The advantage of the analytic approach is – provided the data base is available – that the data can be derived without actually implementing the SDL specification on the target machine. Thus, the analysis can be performed much faster than it is possible with the measurement approach.

With the measurement approach, an implementation is generated for each possible target machine. The implementations are executed with the given MSCs. During the execution, the performance data, i.e. the execution cost for the relevant SDL constructs are measured. In a further step, the cost are integrated in the SDL specification.

The analysis phase is left after subsequent redesign of the functional design specification has lead to an SDL specification which satisfies the basic performance requirements. The main design decisions which are actually met during this redesign cycle deal with the introduction of solution-domain parallelism in the functional design specification. In the cycle, the system is prepared – but not configured – to meet the given performance requirements. The most important decisions are concerned with the strategy for the distribution of data, the strategy for the static distribution of code and the strategy for the dynamic distribution of the load. In addition, also strategies to provide fault tolerance, if required, should be decided on in this phase. The design decisions concerning the parallelization strategies need to be integrated manually into the SDL specification. This is because the respective design decisions are mainly concerned with questions dealing with algorithmic features of the functional specification. Thus, automatic support on the specification level would require the transformation from one algorithm to another, which is undecidable in most cases.

Synthesis In the synthesis phase, which covers the implementation design, the major optimization decisions are made. In other words, the alternatives for parallel execution, which have been identified in the analysis and the functional redesign phase, are evaluated, the best alternatives selected and the missing parameters of the respective strategies optimized.

For system synthesis, model-based optimization techniques can be used to decide on the selection of design alternatives which have emerged in previous phases. For example, assume in the previous redesign cycle a certain SDL process has been parallelized. Then, during implementation design it is decided

- where the SDL process is executable, i.e. on which processors the respective code is available, and
- under what circumstances a given input is handled by a specific processor of the set of processors that can process the input, i.e. the dynamic load balancing strategy.

In general, the synthesis decides on

- the (static) distribution of the code, i.e. the distribution of SDL processes,
- the distribution of data, which again reduces to the distribution of SDL processes due to the encapsulation of data in SDL processes,
- the granularity of the processes handled by the operating system,
- the dynamic load balancing strategy and
- the dynamic scheduling strategy.

The decisions are based on the same basic type of information employed by the analysis phase, i.e. the annotated SDL specification, the extended MSCs and additional design constraints. Additional requirements of the requirements specification, which should be considered, are the cost for system development and production of the different design alternatives and the respective time to market of the design alternatives. In principle, a complex goal function may be employed quantifying the corporate goal in respect to the development of the system. As can be easily seen, a large number of parameters influence the function. Note that different from the design decisions made during previous phases, the decisions made here can be automatically integrated into the implementation.

Implementation The implementation phase is in charge of the generation of the actual parallel implementation. The larger part of the code can typically be derived automatically from the SDL specification as provided by the functional design specification. This is done according to the design decisions specified by the implementation design specification as generated by the synthesis phase.

Commercial SDL tools for code generation are available, which handle the static distribution of code and data. Special tools are needed to automatically generate the code which implements the dynamic load balancing strategy and the dynamic scheduling strategy. In addition to the automatically derived part of the implementation, a part of the implementation needs to be hand-coded. To generate a prototype implementation, this should be kept as small as possible.

From the performance viewpoint, only the parts influencing the performance of the system should be implemented at this stage. However, additional code may be needed to take into account the non-ideal properties of physical hardware, which is not taken into consideration by SDL. Also additional code may be needed to support testing and measurement.

Validation The implementation serves two purposes,

- to test and validate the functional behavior of the system and
- to measure the performance figures of the system and to validate them against the performance requirements.

Validation is still needed, but its automization is highly supported by the formal approach. It may lead to feedback to various previous phases of the development cycle. Testing the functional behavior of the system is an important activity of the development process, and the effort to be put into it highly depends on the rigor with which the preceding activities have been performed. However, functional testing as an activity in the development process is rather independent from the performance issues. For this reason, it is not directly covered here. The reader may refer to (Grabowski, 1994) for a detailed discussion of the issue.

The measurement and validation of the performance of the system is the last phase relevant to the performance of the implementation. This is done in a series of steps. First, the implementation is automatically instrumented to allow for the tracing of performance measures. Then, the implementation is executed with the load specified in the requirements specification, i.e. the extended MSCs. During the execution, the relevant performance data are traced with a monitoring system. The data are analyzed and checked against the performance requirements. In addition, measurements can also be used to validate or update the performance data base employed during the analysis phase. Instrumentation, monitoring and performance analysis can be supported by tools.

4 THE DO-IT TOOLBOX

In order to support the automization of the design and implementation process for parallel systems in the context of SDL and MSCs, the DO-IT toolbox (Design and Optimization – Intelli-

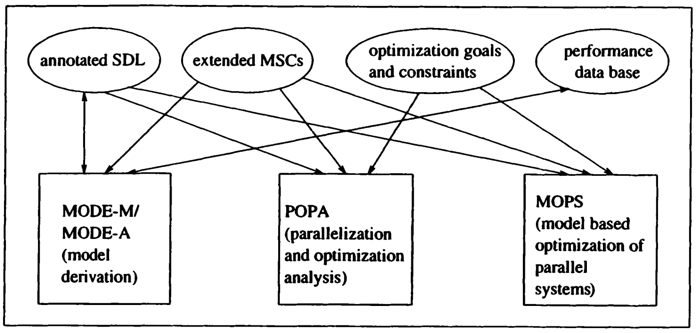

annotated SDL
extended MSCs
optimization goals and constraints
performance data base
MODE-M/ MODE-A (model derivation)
POPA (parallelization and optimization analysis)
MOPS (model based optimization of parallel systems)

Figure 3 The DO-IT toolbox

gent Toolbox) depicted in figure 3, has been devised. The DO-IT toolbox consists of three main components, complementing the commercially available tools for SDL and MSCs.

The analysis phase is supported by three tools, the MODE-A and MODE-M tools to derive performance data for a SDL specification on a given machine and the POPA tool, a tool to analyze whether the performance requirements and the design constraints provided in the requirements specification can be met by the given SDL specification.

MODE-M The MODE-M tool (MOdel DErivation by Measurements) supports the annotation of SDL specifications with performance data. The performance data are derived by measuring the execution times of the SDL specification on a specific machine. As already outlined in the previous section, the term "machine" denotes the combination of hardware and system software that executes the SDL constructs. Thus, the computation and communication cost with which the SDL constructs are annotated reflect the cost of execution of the construct with the given system software on a specific hardware unit. Note that the cost also depend on the code generator and compiler that are used. Thus, each combination of code generator, compiler, system software and hardware constitutes a separate machine. In case several machines are at the disposal of the design process, each SDL construct is annotated with a vector where each element reflects the cost of the SDL construct on a specific machine.

The derivation of the performance data with MODE-M is performed in a series of steps, itself involving a set of tools. First, the given SDL specification is automatically instrumented, i.e. additional instructions are integrated into the code that record start and end time of the relevant SDL constructs. The instrumentation is controlled by MSCs. Our current instrumentation tool performs the automatic instrumentation of C code (Dauphin, Dulz and Lemmen, 1995). However, with support of the code generator for SDL, e.g. as provided by the SDT code generator, the respective instructions can be directly integrated into the SDL specification. This eliminates the need to associate each SDL construct with the respective parts in the C code.

After instrumentation, the code is translated and executed on the target hardware. During execution, the execution times are traced. The execution of the code is controlled by the input signals specified in the MSCs. As a result, only those parts of the SDL specification are typically executed for which a respective MSC exists. Thus, no performance data are traced for the remaining parts of the SDL specification. However, this is not a problem as long as the performance relevant parts of the SDL specifications are covered by MSCs, what is typically the case.

The monitoring of the system and the recording of the traces is either done in software or with the ZM4 hybrid monitoring system (Dauphin et al, 1994). The ZM4 allows for the monitoring of parallel as well as distributed systems and supports a wide range of hardware interfaces. For the analysis of the traces, the SIMPLE analysis tools (Dauphin et al, 1994) and (Hofmann et al, 1994) are used. An additional tool is needed that supports the back annotation of the SDL specification with the measured performance data.

MODE-A Instead of the derivation of the performance data by means of measurements with MODE-M, the analytical modeling tool MODE-A (MOdel DErivation – Analytic approach) may be employed. Central component of this approach is the performance data base. It specifies for a set of relevant machines the performance data of the SDL constructs. Provided that the data base provides the performance data for the required machines, the annotation of the SDL specification can be done quickly without the need to actually implement and execute the SDL specification. The performance data for a particular machine in the data base can either be estimated based on performance data available for a comparable machine or based on measurements previously derived by MODE-M.

POPA The POPA tool (Parallelization and OPtimization Analysis) is the third tool supporting the performance analysis. It provides feedback to the functional design indicating at which parts of the SDL specification further parallelization or optimization is needed to meet the performance requirements of the system. POPA derives its guidelines based on the performance requirements, i.e. the load on the system specified with the extended MSCs, the design constraints and the annotated SDL specification. POPA supports a path analysis and throughput analysis.

The path analysis searches for the critical paths in the SDL specification. The search is based on the extended MSCs given by the requirements specification. POPA checks for each MSC whether the length of its critical path is within the limits specified in the requirements specification.

The throughput analysis computes the load for each SDL process or process instance that results from the load imposed on the system as defined by the extended MSCs in the requirements specification. It checks whether the resulting load can be handled by the machines available, and whether the capacity of the interconnection network that connects the machines, is sufficient. A first implementation based on interworkings – a synchronous variant of MSCs developed by Philips Communication Industries – is described in (Hoppmann, 1993).

MOPS During the functional design and analysis cycle, the SDL specification has been subsequently prepared and optimized to meet the performance requirements. The next phase, called the synthesis or implementation design phase, is concerned with the actual mapping of the given SDL specification on the parallel machine. This is supported by the model based optimization tool MOPS (Model based Optimization of Parallel Systems). The tool decides the major design decisions concerning the static as well as dynamic aspects of the implementation. The design decisions comprise

- the static mapping of the code on the machines, i.e. the decision on which processor a specific SDL process can be executed,
- the granularity of the processes handled by the operating system or the runtime environment,
- the dynamic load balancing strategy for the SDL processes for which more than one instance may exist,
- the dynamic scheduling strategy, e.g. the priorities of the processes, and

- the selection of the most appropriate combination of (1) hardware, (2) system software and (3) code generator for each SDL entity.

The design decisions are computed based on the same information on which the POPA tool is based. However, the MOPS tool is much more sophisticated than the POPA tool since it has to come up with an actual solution to the design problem which comprises a whole set of NP-hard optimization problems, not just an analysis.

Several optimization algorithms have been implemented to compute a part of the design decisions relevant to configurable message passing systems. A class of algorithms is based on the BBU algorithm (Mitschele-Thiel, 1993). They compute the mapping of the code on the machines, the schedule on each machine, and the interconnection network connecting the machines, assuming each machine has a limited number of links that can be freely connected to other machines, e.g. as it is the case with transputer networks. The algorithms employ various heuristics to prune the enormous search space (Mitschele-Thiel, 1994). In addition, a first algorithm based on clustering has been implemented to compute the mapping and the scheduling strategy in case the network topology is fixed (Haidt, 1994). All the algorithms optimize the response time of the system under given throughput constraints. The algorithms cover the design decisions described above, with the exception of dynamic load balancing and support for more than one type of machine. Thus, currently it is assumed that the SDL specification is implemented on a homogeneous parallel architecture and that only one code generator and one type of system software is available.

In order to incorporate all the design decisions, the development of a tool based on genetic algorithms is underway. A first prototype based on the MPGA package (Schwehm, 1993) has shown promising results (Schwehm and Walter, 1994).

The implementation phase for SDL systems is sufficiently supported by commercial tools. A problem, however, is the lack of the ability of the tools of different providers to interoperate. Thus, different interfaces are needed to support the (semi-)automatic implementation of the design decisions made by MOPS.

5 SUMMARY

In the paper a methodology for the integration of performance issues in the development process of parallel SDL systems is described. The methodology is based on two key concepts, namely

- the formal specification of performance requirements and issues related to it and
- the early integration of performance relevant design decisions in the development cycle.

The proposed methodology is supported by the DO-IT toolbox, complementing the commercially available tools for SDL and MSCs.

The goal of the DO-IT toolbox is to support

- the automatic derivation of the performance data incurred with the execution of the SDL specification on specific machines,
- the analysis of the SDL specification for performance bottlenecks and
- the synthesis of the system, providing algorithms to compute the major design decisions relevant to the implementation of SDL specifications on parallel systems.

Central to the early integration of performance issues in the design process are the extension of MSCs to formally specify performance requirements and the annotation of SDL specifications with the performance data associated with the SDL constructs.

REFERENCES

R. Braek, O. Haugen. (1993) Engineering Real Time Systems – An object-oriented methodology using SDL. BCS Practitioner Series, Prentice Hall.

P. Dauphin, R. Hofmann, R. Klar, B. Mohr, A. Quick, M. Siegle, F. Sötz. (1994) ZM4/SIMPLE: a General Approach to Performance-Measurement and -Evaluation of Distributed Systems. Readings in Distributed Computing Systems, Ed. T.L. Casavant, M. Singhal, IEEE Computer Society Press.

P. Dauphin, W. Dulz, F. Lemmen. (1995) Specification-driven Performance Monitoring of SDL/MSC-specified Protocols. Proc. 8th Int. Workshop on Protocol Test Systems, A. Cavalli, S. Budkowski (Ed.).

J. Grabowski. (1994) Test Case Generation and Test Case Specification with Message Sequence Charts, Ph.D. Thesis, Universität Bern, Selbstverlag, Bern.

A. Haidt. (1994) Entwurf und Realisierung eines Clustering-Verfahrens zur Optimierung der Schedule und der Topologie von parallelen Transputeranwendungen. Diplomarbeit, Universität Erlangen-Nürnberg, IMMD VII.

R. Hofmann, R. Klar, B. Mohr, A. Quick, M. Siegle. (1994) Distributed Performance Monitoring: Methods, Tools and Applications. IEEE Transactions on Parallel and Distributed Systems, 5(6).

K. Hoppmann. (1993) Lastanalysator für hierarchische Signalisiernetze. Diplomarbeit, Universität Erlangen-Nürnberg, IMMD VII.

ITU-T. (1993) Z.100, Specification and description language (SDL). ITU.

ITU-T. (1993a) Z.100, Appendix I. ITU, SDL Methodology Guidelines. ITU.

ITU-T. (1993b) Z.120, Message Sequence Chart. ITU.

A. Mitschele-Thiel. (1993) Automatic Configuration and Optimization of Parallel Transputer Applications. Transputer Applications and Systems '93, R. Grebe et al. (Editors), vol. 2, IOS Press.

A. Mitschele-Thiel, K. Dussa-Zieger. (1994) Near-Optimal Compile-Time Scheduling and Configuration of Parallel Systems. Proc. of the 1994 World Transputer Congress, Lake Como, Italy, IOS Press.

A. Olsen, O. Faergemand, B. Moller-Pedersen, R. Reed, J.R.W. Smith. (1994) Systems Engineering Using SDL-92. North Holland.

M. Schwehm. (1993) A Massively Parallel Genetic Algorithm on the MasPar MP-1. Proc. Int. Conf. on Artificial Neural Nets and Genetic Algorithms (ANNGA '93), Innsbruck, Austria, Springer-Verlag.

M. Schwehm, T. Walter. (1994) Mapping and Scheduling by Genetic Algorithms, Parallel Processing: CONPAR'94–VAPP VI, Third Joint Int. Conf. Vector and Parallel Processing, Lecture Notes in Computer Science 854, Springer-Verlag.

Telelogic Malmö AB. (1995) SDT 3.0 User's Guide, SDT 3.0 Reference Manual.

Verilog. (1994) GEODE – Technical Presentation.

3
Designing and implementing complex systems with agents

P. Marcenac(1), S. Giroux(2), J.R. Grasso(3)
(1) IREMIA, University of La Reunion BP 7151, 97715 St Denis Messag. Cedex 9, La Reunion, France. Tel: (+262) 93-82-84 Fax: (+262) 93-82-60 email: marcenac@univ-reunion.fr
(2) Tele Universite, 1001 Rue Sherbrooke Est, 2eme etage, Montreal, Quebec, Canada H2X 3M4. Tel: (+514) 343-6447 Fax: 522-3608 email: sgiroux@teluq.uquebec.ca
(3) LGIT-IRGM, Observatoire de Grenoble, BP 53X 38041 Grenoble, France. Tel: (+33) 76-51-45-17 Fax: 76-51-44-22 email: grasso@lgit.observ-gr.fr

Abstract

This paper describes an ongoing research in the Geomas project, initially intended to study applications of agent technology in complex systems. A complex system can be defined as a system in which behavior is bad-understood and designing such systems then requires specific considerations, justifying the need of the agent paradigm, when no other solutions could be found in an efficient way. The complex system tackled in this paper to illustrate our purposes is the prediction of volcano eruptions. Through the presentation of a simulation application for volcano phenomena, this paper focus on a software engineering approach to agent modelling in simulation. To address such issues, the paper describes an agent architecture through of as software engineering models of agents. A structural approach of the designing task is introduced by 1. conducting a top-down analysis to look for autonomous agents; 2. identifying internal behaviors, interaction processes and evolving facilities of each agent; and 3. looking at the emergence of the global behavior.

The second part of the paper presents some elements of the implementation. ReActalk, an agent-oriented platform, has been chosen as a basis shell to develop simulation applications. ReActalk is built as successive layers developed upon Smalltalk-80, and supports large mechanisms of implementation for both individual agents and global system. It provides a safe combination of passive objects (Smalltalk classes) and actors, bringing serious advantages for the implementation of agents and societies in simulation applications.

Keywords

Agent oriented design, software engineering model of agents, MultiAgent Systems, Simulation Applications, Smalltalk-80.

1 INTRODUCTION

As described in (Bond and al, 1991), different problems types are tacked successfully with MultiAgent Systems. This paper presents interesting results of the Geomas* research project held in the University of La Reunion (France). The main issue tackled in the project is the design and the implementation of distributed complex systems that will satisfy properties of MultiAgent Systems (Marcenac, 1995). The paper describes a paradigm to support the design of such systems, and introduces some key issues associated with the representation of software components in complex systems.

More particularly, in order to explore the emergence of complex behaviors and to derive laws and predicable macro-behaviors out of micro-behaviors, we are conducting a MultiAgent modelling and simulation of the volcano of La Fournaise in La Reunion island, one of the most active volcano of the Earth. The aim of the simulation is to observe the global behavior to look for the number of eruptions according to their volume. To try to understand the complex behavior of the volcano, a computational model with communicating agents is then considered, the result of the system being the emergence of a global solution through the study of local magma pressures. This MultiAgent modelling helps us to investigate such an approach, allowing to integrate partial results in a same frame (Grasso and al, 1995).

This work particularly focus on modelling real world components through agents, and points out what we perceive to be the most important issues in the design of MultiAgent systems for simulation of complex systems. To ease the designer's task, a method based on the *role* of both agents and MultiAgent System is proposed. It describes a structural approach of the designing task by:

1. Conducting a top-down analysis to identify autonomous components of the real world to be integrated as agents, according to the philosophy of what could (or should) be an agent for simulation needs. This part of the work explores the real world by looking for roles played by components in the assumed global mechanism. Such components will then become agents if satisfying agent's principles (according to Gasser and Hunhs, 1987).
2. Identifying characteristics and local goals of the agent and clearly separating *internal behaviors*, *communication* processes and *evolving* facilities of the agent, which we perceive as the most important parts of an agent in complex systems modelling.
3. Identifying the global goal of the whole system, and giving it the role of the society of agents. The society of agents represents the whole system and expresses the emergence of the global behavior.

This methodology involves two complementary studies: first, designing the whole system by looking for the emergence of a global watching behavior, and second, designing each agent, by identifying characteristics to be represented, and internal mechanisms required to describe local behaviors of the agent (Leman and al, 1994).

A first prototype, Geomas-V1, has been developed with ReActalk, an agent-oriented platform enriched to develop simulation applications (Giroux and Senteni, 1991). ReActalk is built as successive layers developed upon Smalltalk-80 and supports large mechanisms of implementation for both individual agents and global system. Geomas V1 was built to validate the designing process and the agent architecture, and for that, low degree of complexity was introduced. We choose to represent sample materials, such as homogeneous rocks and magma feeders, with very sample physical laws to validate the approach.

However, this paper is going further. It keeps a formal view of the way to build such systems and proposes an agent model and architecture for simulation applications. We consider an agent

* Acronym for Geophysics and multiagent systems.

model as the description of the concept of agent (internal behaviors, communication processes and evolving facilities), and an agent architecture as the organization of the society and the understanding of the emergence to explain and formalise the sequence of actions that will achieve the desired goal. This approach leads to a richer semantic in the description of agents, and in a more general way, in the description of the whole system. It allows a better software incrementality, leading to an evolutionary design process of distributed agent systems, consisting of a stepwise development and increment driven.

This paper is divided into two main sections. Section 2 is the core of the paper, and discusses of: .
1. How to design agents and MultiAgent Systems intended to simulate complex natural phenomena, and to derive the emergence of a global behavior from local interactions?
2. How to formalize these concepts in an agent model and architecture?

Section 3 presents some elements of the implementation. Modelling complex systems needs specific requirements, such as modelling the system with a variable size for instance. Programming these features can be done in ReActalk in several ways. As, from a software engineering point of view, the project aims at providing a better incrementality through reusable components, it is common to isolate those which seem to be standard in simulation applications.

Finally, section 4 draws up a report of interesting results and points out future researches.

2 DESIGNING A COMPLEX SYSTEM WITH AGENTS

This section presents conceptual considerations of distributed systems for simulation. It begins by briefly presenting the domain and justifying the agent approach. How such systems could be designed is then investigated. Finally, to tackle this task, an agent-model is proposed.

2.1 Agents and volcano modelling

The volcano is characterized by a very complex structure, which does not follow global physical laws. However, it is driven by three main properties: 1. non predicable property, a small external perturbation could generate a large-scale phenomena; 2. small external perturbations driven; and 3. scale invariance, describing the repetition of watching behaviors during time (Grasso and Bachèlery, 1995).
So the global and watching behavior of the volcano is completely determinist, because it is submitted to a well-determined law (third property), but at the same time, is driven by a non predicable behavior. The combination of these two properties defines a complex and chaotic real world. The volcano is then considered as a global system in which underlying dynamically processes are distributed.

Object-oriented techniques do not provide satisfactory results when modelling such complex worlds. One of the main reasons of this failure is that considering a complex system as a unique entity, and eruptions as the results of the execution, does not allow to describe enough semantics to provide satisfying results. In this kind of centralized program, intrinsic mechanisms could not be deeply taken into account, leading to an insufficient description of the real world. To better understand how does a complex system work, the approach is to consider the program behavior as the result of a set of interactions between smaller and independent agents.

The global behavior of the system is then based on matching the post-conditions of two kinds of local actions: first, reactive actions, in response to external solicitations, and second internal actions, in response to the evolving of the real world component during time. So, the global behavior is evolving during time to take into account internal or external perturbations, and do not have to wait for any global result to do so. It appears then that any computational model could not provide any satisfactory results without autonomous facilities, justifying the need of agents.

2.2 How to design agents

The method proposed is based on a top-down approach. In such a way, the real world is cut in tiny and independent pieces, each one playing a *role* for the application. At a first glance, the method looks like object-oriented methods such as (Coad and Yourdon, 1991), but the designer has to take care of the autonomy of software components, and the 'taskability', i.e. at what level of complexity the activity to be performed by an agent could be described.

Internal behaviors describing the autonomous life and interactions are modelled at the same time, and are encapsulated together in autonomous agents. Many works have been based on this approach, leading to reactive agents (Brooks, 1989), (Demazeau, 1993), (Drogoul and al, 1991). Recent works have defined hybrid architectures of agents (Woolridge and Jennings, 1994), (Muller and al, 1995). Our method of conceptual design of autonomous agents follows this idea, except that, because of the complexity of the real world considered, cognitive agents are described, including more sophisticated protocols. It adopts an external perspective from which to look at agent contents, which relates to L. Gasser's ideas (Gasser, 1990), where agents organisation is treated as aggregated local components.

The whole approach consists first in analyzing the real world to identify autonomous components of the real world to be integrated as agents, then designing each agent separately by looking at internal behaviors, communication processes and evolving facilities and finally designing the whole system. The whole system is then viewed as a society of agents and is responsible of the emergence of a global behavior.

Illustration of the method in the case of the volcano complex system

The first step of the work aims at determining what the simulation should provide in output by identifying the role of the whole system, and what are the roles played by any software components in the system.

. Role of the simulation application: Let remind the aim of our simulation: to observe the global behavior to look for the number of eruptions according to their volume. The role of the corresponding MultiAgent System is then to get knowledge on eruptions which could appear. Inputs of the system consist in magma injections controlled by local components (agents) and outputs consist in measuring the amount of magma ejected from an eruption.
. Looking at agents in terms of roles played by real world components: The internal structure of the Piton de la Fournaise is complex and bad-known, but all researches made during the fifteen past years are converging: the volcano is a network of magma feeders. Elements composing the network are often called 'magma lens', as the network itself is called 'surface tank'. Magma feeders (lens) are inter-connected in a continuous or temporally way, and are separated by a matrix of rocks coming from previous eruptions. However, one can note that magma arrivals are selectively produced; this would reinforce the independent nature of the surface tank, because it works lost of the time in autonomous regulation.

So, lens could be considered as isolated with their own behavior: according to their size, their shape, the magma, and collecting rocks properties, they will react to internal or external perturbations. Internal perturbations are due to magma crystallization inside the lens causing overpressures, and external ones are due to overpressures coming from anywhere else in the volcano. From this analysis, both lens and rocks are playing a defined role in eruption mechanisms, and have then to be modelled as agents in the system.

2.3 The agent model

The Geomas system includes planner-based agents, each one representing a role. An agent is bearing semantics on what it has to do for the whole global system. However, the design, implementation and assessment of MultiAgent Systems for simulation raise many specific issues. More particularly, in simulation environments, three main properties should be studied for each agent: *interaction*, *behavior* and *evolving*.

. Interaction is a basis mechanism when working with agents, it allows to consider a global result as emerging from exchanges between agents. As it is the case in many domains, each agent in a complex system models a natural component of the real world. This agent is assumed to evolve in a specific environment and play a role in its surroundings. So modelling interactions in the system is fundamental for simulation purposes.

. The behavior part of an agent includes what the agent is supposed to do during his life. The behavior can be internal, by working by his own (i.e. without any external solicitations) or external (i.e. when receiving external solicitations). This behavior describes the agent autonomy. In complex systems, each component is assumed to be independent, and could work without external solicitations (see the case of a lens for instance).

. Finally, the last property which should be carefully studied, is the agent capability to evolve during time. Each real world component is subject to modifications. These modifications affect both data and behaviors of the agent, and depend on what was made by the agent during his life.

Matching the model: example

For the volcano simulation, two agents, lens and rock, are designed in such a way:

. Lens: one of the main issue in designing a lens is to model a running fluid flow (magma injection for instance). Indeed, if a volume of magma V has to be transferred to a lens, this one can not be programmed to receive the total amount in one time, because it could not be able to compute its back pressure again and to adjust its behavior during the fluid flow. So the input volume to transfer will be discretized in multiple ΔVs. Performing then a low volume item ΔV at one time results in a dynamic updating of the local pressure and perhaps a different reaction to the next ΔV to perform. From a programming point of view, ΔVs represent time units required to compute again the internal pressure of a lens between two arrivals, i.e. execute the appropriate method in the agent lens.

. Rock: the behavior of a rock agent is expressed by its resistance to magma pressures. The interaction consists in propagating pressures to his neighboring components, and the evolving is linked with the internal change of structure when magma penetrates, breaking the structure of the rock. To balance this behavior, the value of the resistance can be randomly chosen between two given limits.

2.4 The agent architecture and the emergence of a global behavior

The agent architecture describes how agents are organized to form a society. In Geomas, the society is organized as an *ecosystem*, driving all agents to satisfy the desired role and provide an external behavior. The architecture of the ecosystem includes:

. The role of the whole system, expressed through the external behavior, and driven by input and output parameters.

. Knowledge about the structure of agents composing the system (such an organization is represented in Geomas with a network, where agents correspond to nodes and interaction possibilities to edges. The number of agents determining the network size is a global parameter.

. A filtering process, which is responsible of localising which agent will be involved to be consistent with the current goal of the ecosystem.

Matching the architecture: example

The role of the MultiAgent system implementing the volcano simulation is to provide a global behavior allowing to count the number of eruptions according to their volume. Inputs of the system constitute magma injections controlled by local components (agents), while outputs measure the amount of magma ejected from an eruption:

. Example of input parameter: The choice of the injection mode of each lens for a simulation is a global parameter of the system. The injection mode can be continuous or discrete. In continuous mode, magma could be injected in a lens even if the network is not stable. In discrete mode, the network has to be stabilized before a new injection will be applied again.

. Example of output parameter: Two ways to collect eruptions could be envisaged when an injection is applied as input in a lens: counting the amount of ejected magma together for all eruptions, or counting one eruption and the magma amount ejected each time an eruption appears.

. Knowledge about structure: the number of agents determining the size of the network sets the number of lens and rocks for a specific simulation. In addition, the network can be chosen in a loaded initial state or unloaded. When loaded, each initial pressure of a lens will be initialized by a value which is just under the resistance of the rock, which signifies that the volcano is in a critical state. The simulation will then quickly provide eruptions.

Another parameter which constitutes a very interesting feature, is the number of connections between agents. It addresses the way to design a real world as a three dimensional network. When the simulation begins to run, each agent composing the society (rocks and lens) establishes his accointances (neighbors) to be able to communicate with other agents in an asynchronous way. This parameter determines what we call the *degree of communication* of the agents in the system. As connections are determining the ability of an agent to communicate, a part of the communication protocol is then defined by setting this parameter. So the number of connections describes the influence an agent could apply on another and defines a spatial and geometric disposition of agents in the universe. A sample heuristic to implement the degree of communication is to consider a 'corona' in the plan, as illustrated in Figure 1:

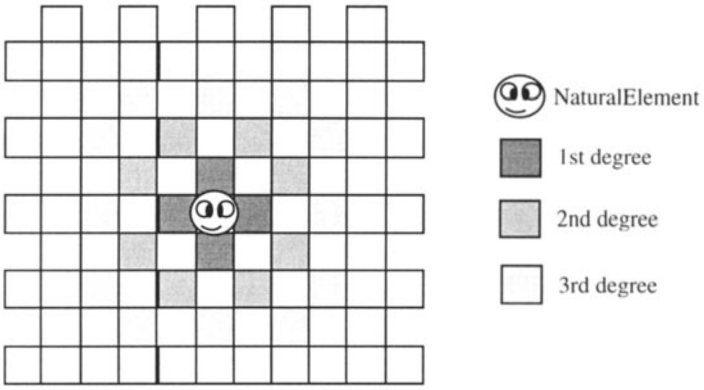


Figure 1 Implementation of a degree of communication.

Finally, note that modelling the volcano as a Cartesian plan wrongly induces a plane view of the edifice. However, it is then possible to abstract this projection, by increasing the degree of communication. This point of view allows a modelling of the real world in three dimensions (3D), by considering 3D connections in a network, as shown in Figure 2:

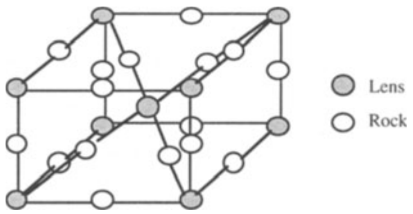


Figure 2 3D structure of the edifice.

The next section gives some details on the implementation of the system.

3 SOME ELEMENTS OF THE IMPLEMENTATION

3.1 Overview

The whole application described in section 2, Geomas V1, has been implemented as a prototype, using an agent-oriented platform named ReActalk (Giroux and Senteni, 1991), (Giroux, 1995). ReActalk is an open environment which provides large adaptive mechanisms for both individual agent and global system.

ReActalk proposes a reflective MultiAgent platform. ReActalk is based on Actalk (Briot, 1989). Actalk is an actor platform for the study of actor paradigms within Smalltalk-80. Indeed, Actalk is a minimal actor system (Agha, 1986), built on top of Smalltalk-80, and designed in order to provide a framework for the study and the exploration of actor languages. The conceptual notion of autonomous agent is then implemented and driven by such actors. When an agent is created, a meta-agent is given to it. This meta-agent acts as a private interpreter and is itself a society of agents. Seeing the meta-level as an organized society leads to an environment in which different mechanisms inherent to the object and actor paradigms can be easily used and integrated (asynchronous and synchronous messages passing, behaviors with and without dynamically evolving...). This environment provides a safe combination of passive objects (Smalltalk classes) and actors and represents a serious advantage for the implementation of agents and societies in simulation applications. Figure 3 illustrates such a platform as a basis tool for the development of agent-oriented simulation applications:

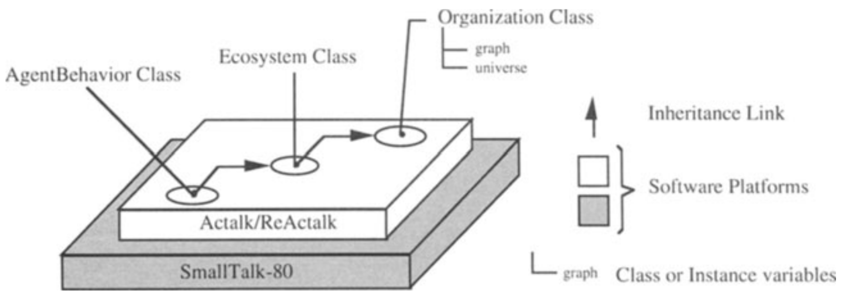


Figure 3 ReActalk, as a basis tool to develop agent-oriented simulation applications.

The notion of autonomous agent is implemented in the *AgentBehavior* class. Each agent is moving in a global entity representing the society of agents, and forming the *ecosystem*. An ecosystem is responsible of all agents it is composed of, and is associated with 1. a structure *to organize* the society; and 2. a structure *to dynamically build and manage* the society.

1. Derived from the ecosystem is the *organization* class which allows to organize the society as a graph. A graph is defined by a set of nodes representing the associated agents composing the society, and a set of edges, one edge between two agents representing a communicating possibility at running step. This graph is built from the location of agents the ones compared with the others. So, the agent's position and the neighborhood determine a matrix in which each agent has his own collection of neighbors.
2. To introduce the management of the society, ReActalk proposes a well-adapted structure for this kind of modelling: the *CartesianPlan* class. By making an instance of the CartesianPlan class explicit in the organization class, the society of agents is managed in an efficient and dynamically way. This instance, named *universe*, is illustrated in Figure 4:

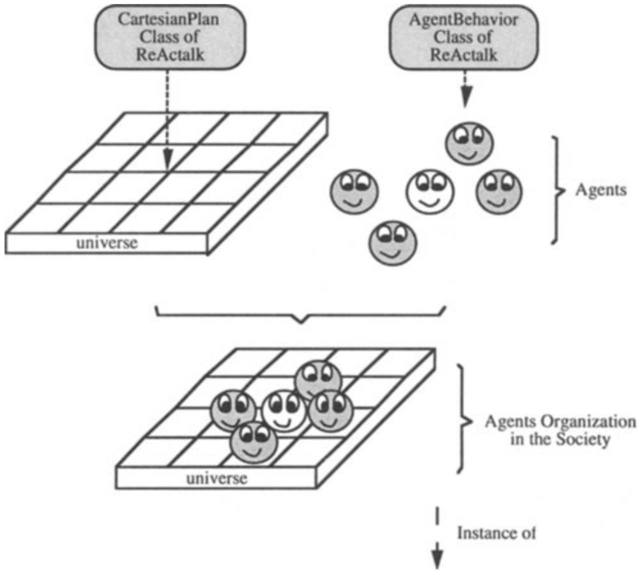

CartesianPlan Class of ReActalk
AgentBehavior Class of ReActalk
universe
Agents
universe
Agents Organization in the Society
Instance of

Figure 4 Organization of agents in a cartesian plan within ReActalk.

In summary, when developing and experimenting with agents, the programmer is connected to the ReActalk environment. Each agent will be defined as a ReActalk agent, and each agent is a member of an organization described as a graph.

3.2 Implementing individual agents

As the first step of the designing, two autonomous components of the real world have been identified for the role they are supposed to play in eruptions mechanisms: 'lens' and 'rock'. A lens describes an amount of magma exerting pressure on its surroundings, i.e. rocks. It is characterized by an internal pressure and an amount of magma. Its behavior is mainly described by geophysical laws such as back pressure laws, indicating how to compute, between two arrivals of ΔVs, the new value of the pressure. Three back pressures laws are implemented (constant, linear and gaussian) and can be switched at each simulation. As internal parameters driving eruptions are not well-known, this constitutes an important facility of the system.

A rock is characterizing a resistance to the pressure of the lens, and its behavior is quite sample at the moment: depending of the value of the resistance, a rock could let drain the magma across or not. This sample modelling is not realistic when looking for internal structures of the volcano. As the aim of the system is to provide a basis for studying eruptions, and the link with the geologic structure of the volcano has not yet been identified, the complexity of a rock does not matter. However, taking account the dynamic evolving of the rock and modelling its complexity are under investigations in Geomas V2.

However, one must keep in mind that lens and rock agents will evolve in a global entity representing their society, the ecosystem. As we seen it before, each agent in the ecosystem has to know his current position for easier identification, and his neighborhood for knowledge on which agents he could communicate with. With that aim, two slots are added, *position* and *neighbors*, plus one for the *membership* of the ecosystem.

Furthermore, these characteristics are not specific to a rock and a lens; and by the generalization mechanism, are isolated in an abstracted class, *NaturalElement*, between the AgentBehavior class of ReActalk and application classes. This kind of designing which looks at roles and intrinsic properties of an agent and a society of agents, leads to develop an abstracted level of reusable components. Figure 5 illustrates such a feature:

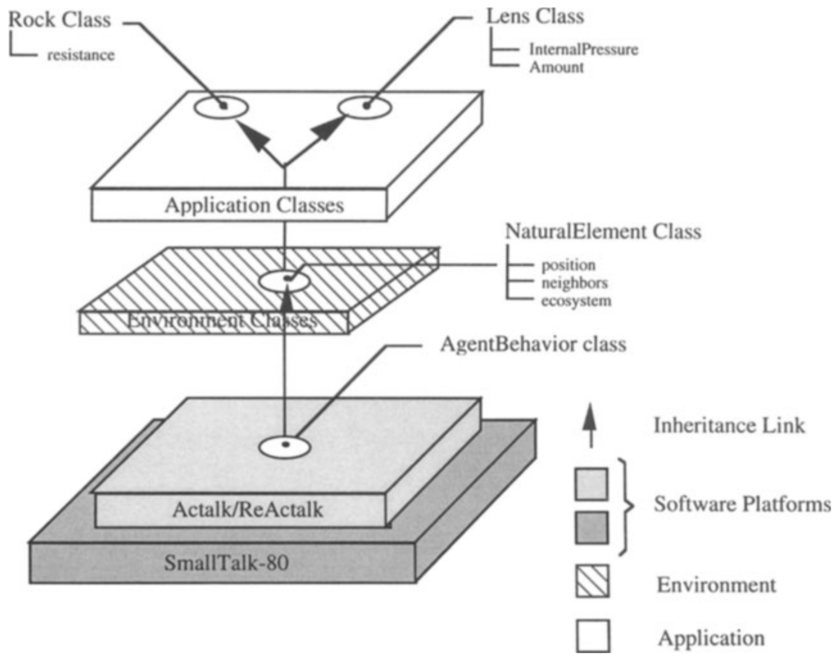


Figure 5 Different Software levels upon Smalltalk-80 to develop simulation applications.

3.3 Implementation of the volcano structure

To implement the structure of the society, a class named *PitonFournaise* has been defined. This class describes the whole structure of the volcano and allows to observe the global behavior.

One can note that in the case of the volcano application, the organization of the society is static; agents do not change position, nor neighbors. As a graph is a quite-well adapted structure in dynamic environments, it is not required in this case. So the PitonFournaise class could also be derived directly from the ecosystem class. This point of view bears a performance advantage while the system runs with several thousands agents in parallel, but is too restrictive to be used anyway.

3.4 Results

The resulting product Geomas V1 has been tested for six months in our team by geophysical researchers. It has been performed more than two hundred simulations by updating input parameters in each agent (Lens, Rocks and PitonFournaise). Each simulation has been run on Sun Sparc Solaris 2, during 8 to 10 hours, with around 7,500 agents working in a parallel way (this number has been assumed to be sufficient to match the real world). Around 33000 eruptions were accounted for each simulation (only 76 are actually registered in real data on the Piton de la Fournaise!).

These simulations have pointed out very interesting and promising results. Real data registered have been compared with simulation results, and some mechanisms have been identified and partially explained. In addition, some of the complex parameters assumed to play a role in eruption mechanisms have been identified too.

4 CONCLUDING REMARKS

In simulation applications, a MultiAgent approach becomes fundamental when tackling complex problems, and when no other solutions could be found in an efficient way. Through the presentation of a simulation application to understand volcano behavior, this paper focus on a designing methodology for MultiAgent Systems and proposes then an agent architecture through of as software engineering models of agents.

Designing a MultiAgent System for simulation purposes can be done with a structural approach, by:

1. Conducting a top-down analysis to identify autonomous components of the real world to be integrated as agent.
2. Identifying characteristics and local goals of the agent and clearly separating internal behaviors, communication processes and evolving facilities of the agent.
3. Identifying the goal of the whole system, this one expressing the emergence of the global behavior.

Finally, to implement these ideas, a specific tool, ReActalk, has been presented as a basis platform for developing simulation applications. Actual works on MultiAgent Systems are laying the foundations of new models of computing and interaction, and this kind of precision enforces the necessity of a well-adapted software development tool, which contains intrinsic properties of the agent-oriented paradigm. ReActalk provides satisfying mechanisms to implement a specific level of agents for simulation purposes and easily implement the architecture. The software complexity could then easily be increased at each stage, by adding more complex protocols as developing more complex software. This approach authorises an incremental development and an evolutionary design process, in which stages consist in expanding an operational system.

The architecture has been studied with the aim of helping software designers to get closer to such goals. One of the next step of the project is to add more complexity in Geomas V1, by taking account the dynamic evolving of the rock and modelling its complexity.

ACKNOWLEDGEMENTS

This project is supported by the IREMIA laboratory of the University of La Réunion and LGIT in Grenoble (France). We would like to thank people who have brought their contribution to that work, and more particularly M.L. Aimelet, S. Calderoni, D. Grosser, F. Lahaie and E. Valcares.

REFERENCES

Agha, G. (1986), *Actors: a model of concurrent computation in distributed systems*, MIT Press, Cambridge, Massachussetts, USA.

Bond, A.H. and Gasser L. (1991), An analysis of problems and research in DAI, in *Readings in Distributed Artificial Intelligence* (Bond and Gasser eds), Morgan Kaufmann Publishers, Inc., San Mateo California, USA, Pages 3-36.

Briot, J.P. (1989), Actalk: a testbed for classifying and designing actor languages in the smalltalk-80 environment, *ECCOP89, European Conference on Object-Oriented Programming*, Cambridge, England.

Brooks, R.A. (1989), A robot that walks: emergent behaviors from a carefully evolved network, *AI memo*, **1091**, Massachusetts institute of Technology, Cambridge, Massachussetts, USA.

Coad, P. and Yourdon, E. (1991), *Object-Oriented Analysis, Object-Oriented Design*, Yourdon Press, Prentice Hall.

Demazeau, Y. (1993), La plate-forme PACO et ses applications, *2ème journée nationale du PRC-IA sur les systèmes multi-agents*, Montpellier, France.

Drogoul, A., Ferber, J. and Jacopin, E. (1991), Viewing cognitive modeling as eco-problem-solving: the pengi experience, *Cahiers du LAFORIA*, Rapport N° 2/91, University of Paris VI, France.

Gasser, L. and Hunhs, M.N. (1987), *Distributed Artificial Intelligence, volume 2*, Pitman, London, UK.

Gasser, L. (1990), Conceptual modeling in distributed artificial intelligence, *Journal of the Japanese Society for Artificial Intelligence*, **5:4**.

Giroux, S. (1995), Agents and actors: a necessary unity, *IJCAI Worshop on agents*, Montreal, Canada.

Giroux, S. and Senteni, A. (1991), ReActalk, a reflective version of Actalk, *OOPSLA'91 Workshop on Metalevel Architecture*.

Grasso, J.R. and Bachèlery, P. (1995), Hierarchical organization as a diagnostic approach to volcano mechanics: validation on Piton de la Fournaise, *Geophysical research Letters*, in press, 1995.

Grasso, J.R., Giroux, S. and Marcenac, P.(1995), A multiagent approach for volcano behavior simulation, *Application of Artificial Intelligence Computing in Geophysics*, International Union of Geodesy and Geophysics, 21st General Assembly, Boulder, Colorado, USA.

Leman, S., Marcenac, P., Aubé, M. and Senteni, A. (1994), Multiagent models for cryptarithmetic problem solving, *CWDAI'94, Canadian Workshop on Distributed Artificial Intelligence*, Banff, Alberta, Canada.

Marcenac, P.(1995), The Geomas project: research report, *IREMIA*, April 1995, 45 Pages.

Müller, J.P., Pischel, M. and Thiel, M. (1995), Modeling reactive behavior in vertically layered agent architectures, in *Intelligent Agents*, (M. Wooldridge, N.R. Jennings Eds), Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, Lecture Notes in Artificial Intelligence, Vol. 890, Pages 261-276.

Wooldridge, M., Jennings, N.R. (1994), Agent theories, architectures, and languages: a survey, in *Intelligent Agents*, (M. Wooldridge, N.R. Jennings Eds), Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, Lecture Notes in Artificial Intelligence, Vol. 890, Pages 1-39.

BIOGRAPHY

P. Marcenac is lecturer at the University of La Reunion (France). He obtained his PhD in 1990, and is involved for five years in multiagent systems. He actually manages the Geomas research project. S. Giroux is researcher at Tele-University in Montreal (Canada). During his PhD and post-doctoral studies, he developed ReActalk, as an open environment to develop MultiAgent Systems. J.R. Grasso is a geophysical researcher in LGIT (France). His main interest is to apply a new paradigm to model volcanoes, based on the agent methodology, in order to capture their internal behaviors.

4

Communications are Everything: A Design Methodology for Fault-Tolerant Concurrent Systems

A.M. Tyrrell
Department of Electronics
University of York, Heslington, York, Y01 5DD, Email: amt@ohm.york.ac.uk

Abstract

Limiting the extent of error propagation when faults occur and localising the subsequent error recovery are crucial elements in the design of fault tolerant parallel processing systems. Both activities are made easier if the designer associates fault tolerance mechanisms with the underlying communications of the system. With this in mind, this paper has investigated the design of such systems, which enforces a design concentrating on the modelling and analysis of interprocess communications providing a better match between the needs of the fault-tolerant mechanisms and the communication structures themselves.

Keywords

Fault-tolerance, concurrent systems, communications, software design.

1 INTRODUCTION

A distributed processing system, comprising a set of discrete processing units, offers the user not only the prospect of increased efficiency and throughput through parallelism, but its inherent redundancy might also be exploited to enhance reliability. To do so requires a properly designed fault tolerance infrastructure which maintains the integrity of the system under fault conditions, in particular communications. This paper describes a design methods which concentrates on the communications within the system, which facilitates the design, placement and implementation of fault-tolerant software mechanisms across a parallel system to ensure safe operations in the presence of faults.

Fault tolerance is often incorporated into a design as a ruggedisation process to protect a process or set of processes regarded as critical to safe system operation (Lee and Anderson 1991). The fault tolerance mechanisms are required to recognise faults by the errors they cause and to prevent error migration from the faulty process to elsewhere in the system, so that error recovery is localised. The extent of the error recovery operation can be limited if the communications structure in the system can be analysed accurately, and a boundary can be identified within the state-space of the distributed system across which error propagation by interprocess communication is impossible; it must include all processes which interact with the function being protected and exclude all processes that do not interact with it. In other words, the state-space of the system has to be partitioned into a hierarchy of atomic actions (Jalote and Campbell 1986). It is then possible to design a distributed error detection and recovery mechanism around the atomic action which ensures that all the processes affected by the fault co-operate in recovery. This localisation of fault tolerance simplifies the design and can help to meet timing constraints in real-time systems (Anderson and Knight 1983).

The design described in this paper concentrates on the communications mechanisms within an application, and within the fault tolerance mechanisms themselves. The design shows how different communication structures help not only in the design of the particular application itself, but more importantly in the design of the fault-tolerant mechanisms protecting the system against faults latent in the system.

2 ATOMIC ACTIONS AND FAULT-TOLERANCE

Firstly, let us consider the crucial role communications play in the operation of fault-tolerant mechanisms in a parallel processing environment. To an external observer the activity of a process is defined by its sequence of external interactions; any internal actions (of which there may be many) can not affect the external observer, at least until the next external interaction. This allows the concept of an atomic action to be derived: the activity of a set of processes is defined as an atomic action if there are no interactions between that set of processes and the rest of the system for the duration of that activity. The extension to hierarchically nested atomic actions is straightforward. These concepts are well-known in distributed transaction processing (Mancini and Shrivastava 1988) from which field many other attributes of atomic actions, such as serialisability, failure atomicity and permanence of effect can be defined.

The process of identifying the atomic actions within a parallel system design brings into clear focus the structure of interprocess interactions and thus the route by which errors might propagate under fault conditions — an obviously crucial aspect in the detection and implementation of the fault tolerant mechanism. All common mechanisms for providing fault tolerance in parallel systems, such as forward error recovery (Randell 1975), N-version programming (Avizienis 1985), conversations (Randell 1975), consensus recovery blocks (Scott et al. 1987) and distributed recovery blocks (Kim and Welch 1989), have to cope with error confinement and achieve this by imposing logic structures 'around' atomic actions.

A generalised fault tolerant mechanism could be considered as a co-ordinated set of recoverable blocks, with one recoverable block in each interacting process, allowing distributed error detection and recovery. The mechanism is bounded by a set of start states (***entry line***), a set of finish states (*exit line*) and two side walls which completely enclose the set of interacting processes which are party to the mechanism, and across which interprocess interactions are prohibited. The structure is indicated diagrammatically in Figure 1. Note that it is the communication pattern that defines the side walls, processes which are interacting are within the side walls (processes R, S and T), processes which do not interact are outside the side walls (processes P and Q).

Two types of communications are illustrated in Figure 1; the lines between the 'recoverable processes' represent the *application* interactions, and are of a consequence of data requirements between the parallel processes. It is these interactions that will define where atomic action exist within the system structure, and thus where fault-tolerant mechanisms should be placed. The second type of communications are those forced upon the application by the fault-tolerant mechanisms. These will typically consist of exchanging data values for voting and/or for comparison, of passing reconfiguration information and signals around the system, and for the recovery of the parallel processes within the fault-tolerant mechanism. This second class of communication would not be present in non fault-tolerant systems, and in many respects should be more secure than the 'normal' application communications.

The entry line defines the start of the atomic action and consists of a co-ordinated set of recovery points for the participating processes. These processes may enter the atomic action asynchronously. The exit line comprises a co-ordinated set of acceptability tests, or voting procedures. Only if all participating processes pass their respective acceptability tests (or the voting procedures are successful) is the mechanism deemed successful and all processes exit, in synchronism, from the action. If any acceptability test is failed, recovery is initiated and processing "passed" to another set of recoverable processes, or set of actions. Thus all processes in the atomic action co-operate in error detection. Note how both synchronous and asynchronous communication structures are present in these mechanisms.

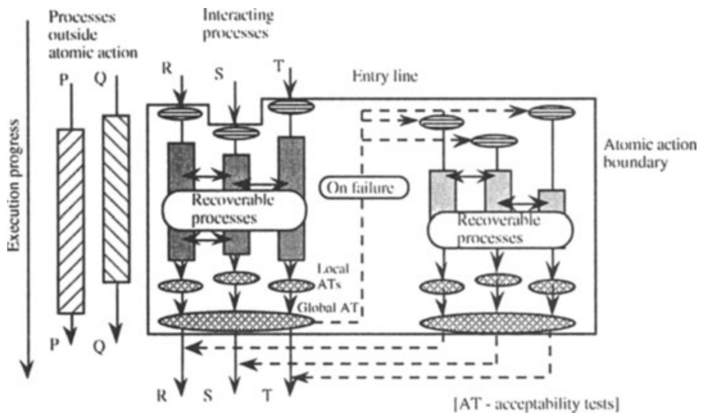


Figure 1. The structure of a fault tolerant mechanism involving processes R, S, and T.

Any attempt to incorporate an entry line and an exit line at arbitrary locations in a concurrent system is unlikely to lead to a properly formed recovery mechanism. It is necessary to identify a boundary within the state space of the complete set of processes across which error propagation by communication is prevented (Tyrrell and Carpenter 1995). Clearly, this boundary will be the boundary of an atomic action, since such a boundary, of necessity, prohibits the passing of information to any process not involved in the atomic action and similarly embraces all interacting processes within the atomic action. Recovery mechanisms can be nested systematically in the same hierarchical fashion as atomic actions. If this duality is not imposed, then should the system attempt to backtrack and recover in response to a fault, progressive collapse by the domino effect (Randell 1975) can occur.

3 FAULT MODEL

It is important at this stage to say a little about the types of faults that can be expected in the systems that are being considered. The fault model for these system comprises of both software and hardware faults.

Hardware Faults :

- dead processor (due to failure of processor or support chips),
- dead interprocessor communication (due to failure of communication hardware),
- erroneous interprocess communication (due to transient fault in processor or communication hardware).

Software Faults :

- differential mode faults (ie. software versions fail independently of each other),
- common mode faults (ie. software versions fail in same manner under the same conditions),
- faults due to difficulty factor (ie. versions fail in different ways under the same system conditions).

While more subtle and complete fault models have been suggested, this fault model provides sufficient ability to give a good idea of the effectiveness of the fault-tolerant mechanisms under consideration.

4 COMMUNICATIONS MODEL

A common communications model used in many fault tolerant systems is that of communicating sequential processes (CSP). This model provides a synchronous non-buffering communications procedure only. While this allows analysis of communications structures, and a effective implementation environment, eg transputers, there are some limitations to this model when used for fault-tolerant mechanism design and implementation. This form of communication is useful for the description of communications that are required between processes that are being forced into synchronous operation at points through their non-synchronous (asynchronous) execution. Problems do occur when implementing and analysing such system designs, when time-outs are introduced to prevent these synchronous communications from allowing a faulty process to stop non-faulty processes.

A more comprehensive suite of communication mechanisms are required if fault-tolerant mechanisms are to be really useful in real applications. Such a communications model has been described by Simpson (Simpson 1994a). This communications model will be used here to design fault-tolerant mechanisms and show how they would be implemented with such a model.

The communications model can be broadly categorised into one of four regions (Simpson 1994b), this is illustrated in Figure 2.

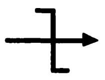
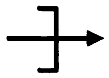
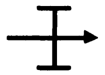
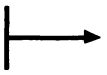
	Destructive Reading *Reader can be held up*	Non Destructive Reading *Reader cannot be held up*
Destructive Writing *Writer cannot be held up*	 SIGNAL *Event Data*	 POOL *Reference Data*
Non Destructive Writing *Writer can be held up*	 CHANNEL *Message Data*	 CONSTANT *Configuration Data*

Figure 2 Communication Model.

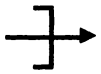
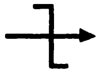
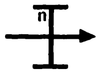
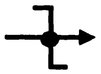
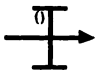
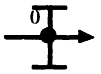
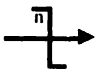
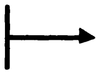
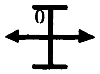
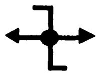
Interaction Function	Symbol	Writer Can Be Held Up	Reader Can Be Held Up
POOL		N	N
SIGNAL		N	Y
CHANNEL/BOUNDED BUFFER		Y	Y
STIMULUS/INTERRUPT		N	Y
RENDEZVOUS		Y	Y
HANDSHAKE		Y	Y
OVERWRITING BUFFER		N	Y
CONSTANT		-	N
REMOTE FUNCTION CALL		Y	Y
REMOTE THREAD INVOCATION		Y	Y

Figure 3 Enhanced Communication Model.

In (Simpson 1994b) these communication models are described as follows: *Pools* allow reference data to be passed from one function to another. This data is retained within the pool where it can be consulted at any time by the reader and refreshed at any time by the writer. *Signals* allow event data to be passed from one function to another. This data can be overwritten at any time by the writer, but can only be actioned once by the reader. The signal is important in real-time systems as it avoids back propagation of temporal interaction effects. *Channels* allow message data to be passed from one function to another. It is normal for channels to have a capacity of more than one, when they become synonymous with a bounded buffer. The message in a channel cannot be lost. *Constants* can be considered as configuration data, and provide a write-once capability.

Note in this model the categories are with respect to destructive reading and writing. Within this simple model, "standard" message passing can be fixed, as can a form of shared data space, asynchronous data, and even global data.

This model can be further enhanced to incorporate system characteristics such as buffer size (shown as n or 0), uni- and bi-directionality (shown by the arrows on lines) and non-data passing (ie stimulus only) indicated by a blob. This enhanced model (Simpson 1994b) is shown in Figure 3.

These enhancements give what is thought as a full model of communication procedures required to describe computer systems. It is noted in (Simpson 1994a) that *signal* and *pool* variants are the most useful for real-time applications. These are difficult to model in synchronous-only models, however this model provides the dynamic characteristics required for these models. We will now go on to show how this communications model can be used to design fault-tolerant mechanisms in a parallel environment.

5 FAULT-TOLERANT DESIGN FOR CONCURRENT SYSTEMS

The use of atomic actions enables many of the problems associated with introducing fault-tolerance into distributed/parallel system to be solved. One of the major problems with these ideas is that in order to identify processes that may be considered atomic actions, the dynamics of the processes and thus the state space of the system must be modelled. The author has successfully achieved such analysis by using Petri-net and GMB graphical methods, and CSP mathematical methods (Carpenter and Tyrrell 1989, Tyrrell and Holding 1986, Tyrrell and Carpenter 1995).

For the introduction of fault-tolerant mechanisms to aid the system designer, there should be provided a set of *framework processes* within which the application program will sit. The structure of these framework processes should be of no concern to the application designer, the only application specifics in its incorporation into the design should be the design of the error detection mechanisms (whether hardware or software, this will always be application dependent), and in the actual placement of the framework process across the distributed/parallel system.

Once the atomic action boundaries have been identified, the chosen framework process (such as, forward error recovery, backward error recovery, and error masking) can then be placed around these safety critical application processes. We will now look at one of these mechanisms, and see how the communications model helps in its design (and in a final implementation).

Enhanced Distributed Recovery Blocks.

The mechanism used is based on distributed recovery blocks (Kim and Welch 1989). It is argued that distributed recovery blocks (DRB) are well suited for real-time control applications since:

- DRB require code versions of graded complexity; a requirement which should easily be satisfied by the plethora of new and classical control theories which are in existence,
- DRB offers distributed operation over a number of redundant processing nodes,
- In the event of a fault DRB dynamically reconfigures the operation of these nodes in order to obtain the maximum possible performance from the hardware available,
- In the event of faults DRB will fail gracefully, always using the highest graded code version available to it,
- DRB relies on acceptance tests, rather than voting, to judge the correctness of results; this is important as voting between alternative control algorithms can be unreliable due to their tendency to produce correct, but different results,

- DRB is proposed as a uniform way of dealing with hardware and software faults; it obviates the need to identify the origin of a fault, which is a costly overhead in terms of time, and has been a major difficulty in real-time computing designs.

DRB are based on the standard method of recovery blocks, Figure 4. The enhancements incorporated within DRB include the concurrent execution of the try blocks over a distributed network of processing nodes and the dynamic reconfiguration of nodal operations in the event of a fault. The systems proposed in this paper takes the basic DRB and introduces extra acceptance tests to reduce the chances of Byzantine type errors and is termed an Enhanced DRB (EDRB), Figure 5. The acceptance test in the EDRB scheme are carried out concurrently on N different nodes. In addition the local database of previous data which is maintained on each node will be exactly the same. The DRB maintains separate databases on each node these are regularly exchanged and compared, thus guaranteeing that they are the same and eliminating the need for any form of roll-back recovery in the event of a detected error. EDRB performs a vote at the start of every iteration to ensure each node is operating with exactly the same data.

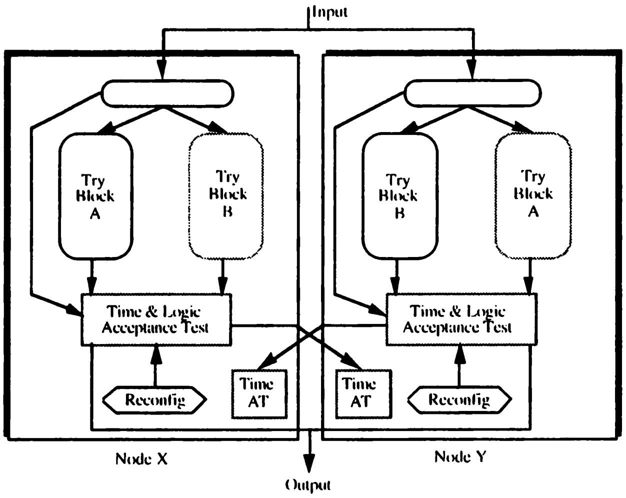


Figure 4 Distributed Recovery Block.

6 COMMUNICATION FAILURES

In an earlier paper on the EDRB (Elphick et al. 1993), a design was reported which in addition to designing the EDRB around real-time applications a number of features were added to the design to help cope with communication failures. When implemented with a CSP model, link procedures were used to detect communication failures (or time-outs), ensuring that other non-failed nodes continued to operate correctly. Reinitialisation procedures were also used to reset failed nodes and allow them to be re-included on the next iteration of the loop (assuming non-permanent hardware faults). The calculation of these time-outs were non-trivial and prone to errors.

These link procedures where very much a consequence of the CSP, synchronous model used. Here it is shown how this new communications model gives a more general solution, allowing particular interactions between parallel processes to be more closely modelled by specific communication structures.

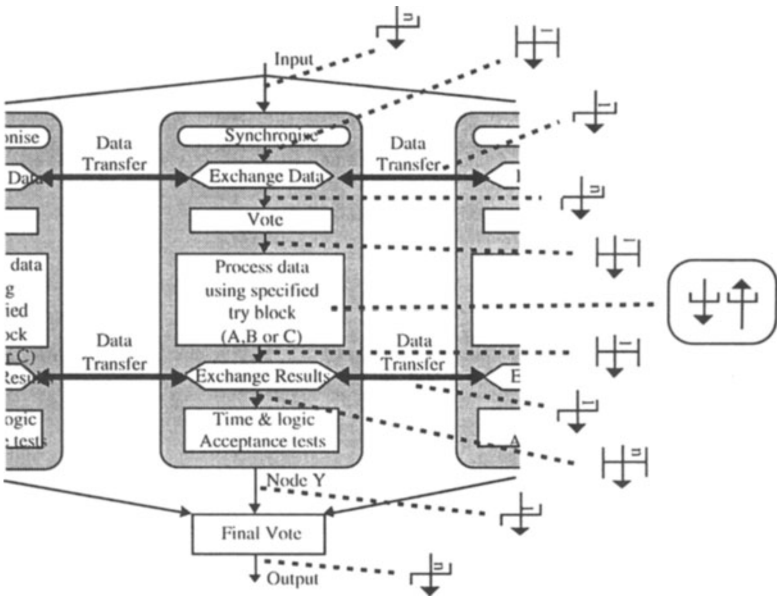


Figure 5 Enhanced Distributed Recovery Block.

Generally, it can be seen in this design that the communications consist of a number of signal and overwriting buffers, and channel or bounded buffer interactions. During the application processing itself, some reference data may be read/written — illustrated by the pool interactions. Indeed some timing information is likely within this process itself, and can be easily specified using this design method. A simple control loop is shown in Figure 6 (Simpson 1994a) illustrating this design method. More detail of this is given in (Simpson 1994a). The choice of whether the interaction should be a signal or a channel is dependant on if the writer should (can) be held up by the reader or not. This is important, for example, when interaction with processes external to the atomic action (ie at the input and output) so that the processes time constraints do not affect the external environment (eg this could be the controlled process). Another example for the use of signals is when the parallel processes are exchanging data for voting and comparison; here we would not wish the writing process to be held up. In certain cases however, it is important that both writer and reader are held up, for example, when the input data is read in and the processing cycle of each process is synchronised.

It appears that by using this new, more general communications model, the system design is better suited to the functions required of it by the particular interactions between the parallel processes. The richer set of communication primitives in this model enables explicit communication structures to be built for specific jobs within the system; in particular in this application, specific for the EDRB. These explicit communication structures force the designer of the system to consider the most applicable structures for the particular interactions. This should produce a design closer to the application, and hopefully a design that is less likely to perform incorrectly in the final implementation. The implementation of such a system should also be easier than an equivalent one using just synchronous communications; Using this richer set of communication structures, many of the timing problems associated with the purely synchronous design disappear allowing a simpler design to be arrived at. Obviously, one has to pay for such an improvement! Many of the problems associated with the purely synchronous design methods are removed by the more "complex" set of mechanisms provided in this new communication set. One could consider that the problems have now been removed to the hardware mechanisms controlling the communications. This assumes that hardware mechanisms are available to implement the different communication structures. It is mentioned in (Simpson 1994b) that chip support for these communication primitives is being designed in the form of a kernel executive chip and a comms executive chip.

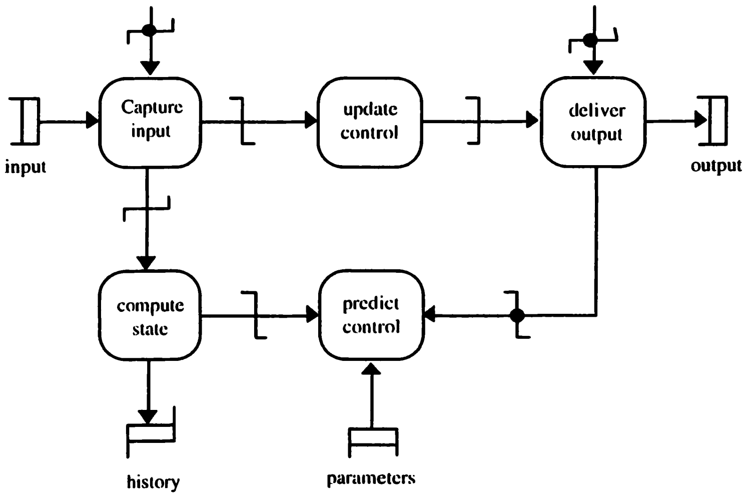


Figure 6 Example Control Process.

7 CONCLUSIONS

This paper has proposed that communications are crucial to the design, and implementation of fault-tolerant mechanisms applied to parallel processing systems. It has shown how a more general model of communications, than that of synchronous communications, can provide better mapping from what is required in fault-tolerant mechanisms. This has been illustrated by a design of a particular fault-tolerant mechanism, but as with the concept of atomic actions, these are general conclusions and should be applied to all fault-tolerant mechanisms designed for operation within a parallel environment.

It has been shown in previous papers that atomic actions should form the basis for fault tolerant mechanisms in a parallel environment; this paper shows how they can be designed and implemented in a systematic, proper fashion. Work is continuing to improve this design method for fault-tolerant mechanisms, as is an implementation of these communication structures by others.

It was proposed in a previous paper (Tyrrell 1994) that a set of design procedures for fault-tolerant distributed/parallel systems could be as follows:

- design a set of application processes,

- model these using an appropriate state space method,
- identify the safety critical functions of the system,
- identify the atomic actions associated with these safety critical processes,
- place the appropriate framework process(es) around these atomic actions,
- design error detection mechanisms for the application in question.

The ideas proposed in this paper would support these design procedures, and enhance them by the introduction of a rich set of communication primitives allowing the mappings from one stage of the design to the next to be achieved easily and naturally.

8 ACKNOWLEDGEMENT

The author would like to thank Geof Carpenter and Hugo Simpson for their ideas on some of the work related in this paper.

9 REFERENCES

Anderson, T. and Knight, J.C. (1983) A framework for software fault tolerance in real-time systems. IEEE Transactions on Software Engineering, **9**, **12**, 355-364.

Avizienis, A. (1985) The N-version approach to fault-tolerant software, IEEE Transactions on Software Engineering, **11**, **12**, 1491-1501.

Carpenter, G.F. and Tyrrell, A.M. (1989) The use of GMB in the design of robust software for distributed systems. Software Engineering Journal, **4**, 268-282.

Elphick, J.R. Patton, R.J. and Tyrrell, A.M. (1993) Enhanced Distributed Recovery Blocks: A Unified Approach for the Design of Safety-Critical Distributed Systems. IEE Colloquium on Safety Critical Distributed Systems, IEE London, Digest No: 1993/189.

Jalote, P. and Campbell, R.H. (1986) Atomic actions for fault tolerance using CSP. IEEE Transactions on Software Engineering, **12**, **1**, 59-68.

Kim, K.H. and Welch, H.O. (1989) Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications. IEEE Transactions on Computing, **38**, **5**, 626-636.

Lee, P.A. and Anderson, T. (1991) Fault Tolerance: Principles and Practice. Springer Verlag.

Mancini, L.V. and Shrivastava, S.K. (1988) Replication within atomic actions and conversations: a case study in fault-tolerance duality. FTCS-19, Chicago, 454-461.

Randell, B. (1975) System Structure for Software Fault Tolerance. IEEE Transactions on Software Engineering, **1**, 220-232.

Scott, R.K. Gault, J.W. and McAllister, D.F. (1987) Fault-tolerant software reliability modelling. IEEE Transactions on Software Engineering, **13**, **5**, 583-592.

Simpson, H.R. (1994a) Temporal Aspects of Real-Time System Design. IEE Colloquium on Methods and Techniques for Real-Time System Development, IEE Press.

Simpson, H.R. (1994b) Architecture for Computer Based Systems. Proceedings of the 1994 Tutorial and Workshop on Systems Engineering of Computer-Based Systems, Stockholm, 70-82.

Tyrrell, A.M. and Holding, D.J. (1986) Design of reliable software in distributed systems using the conversation scheme. IEEE Transactions on Software Engineering, **12**, **7**, 921-928.

Tyrrell, A.M. (1994) The Design of Fault Tolerant, High-Performance Control Systems. IEE Colloquium on High-Performance Computing for Advanced Control, IEE London, Digest No: 1994/241.

Tyrrell, A.M. and Carpenter, G.F. (1995) CSP Methods for Identifying Atomic Actions in the Design of Fault Tolerant Concurrent Systems. IEEE Transactions on Software Engineering, **21**, **7** 629-639.

10 BIOGRAPHY

Dr Tyrrell received a 1st class honours degree in 1982 and a PhD in 1985, both in Electrical and Electronic Engineering. He joined the Electronics Department at York University in April 1990, and was promoted to Senior Lecturer in 1995. Previous to that he was a Senior Lecturer at Coventry Polytechnic. Between August 1987 and August 1988 he was visiting research fellow at Ecole Polytechnic Lausanne Switzerland. His main research interests are in the design of parallel systems, fault tolerant design, software for distributed systems, simulation using parallel computers and real-time systems. In the last five years he has published over 60 papers in these areas, and has attracted funds in excess of £500,000.

5

Designing Distributed Multimedia Systems using PARSE

*A.Y. Liu**   *T.S. Chan***   *I. Gorton****

*CaST Lab, School of Computer Science and Engineering,**
University of New South Wales, Kensington, NSW 2052, Australia
tel: +61 -2 385 4019, fax: +61 -2 385 5995
contact email: annaliu@cse.unsw.edu.au

*Division of Radio Physics, CSIRO, Sydney, Australia***
contact email: tchan@rp.csiro.au,

*Division of Information Technology, CSIRO, Sydney, Australia****
contact email: ian.gorton@dit.csiro.au

Abstract

With recent vast improvements in computer hardware, in particular, the processing capacity of multimedia database servers, and high performance of networks, distributed multimedia applications are becoming a reality. This paper presents an object-based approach to the design of distributed multimedia software. In particular, the PARSE methodology for designing parallel and distributed systems is employed. Justification of the object-based approach is given, and an overview of the PARSE process graph notation is presented. A case-study of a video-on-demand application is then presented, and a mapping from the design to an implementation based on Windows NT is described.

Keywords

Distributed system design, multimedia systems, parallel software engineering, PARSE

1 INTRODUCTION

Advances in computer and media technology have enabled the development of high performance multimedia workstations and servers (Jadav.1995), (Taylor.1995). In addition to processing traditional computer data, these workstations are designed to integrate processing of other media types, such as video, image, voice and sound. On another level, the emergence of high-speed, broadband networks such as B-ISDN (Broadband Integrated Services Digital Network) (Minzer.1989) have accelerated the development of highly interactive distributed multimedia systems (Furht.1994). These systems are designed to transport high bandwidth multimedia information across the network, while supporting real-time interactive interfaces.

Applications developed using multimedia technology can benefit significantly from the rich expressive graphical presentation, with the potential to incorporate distributed and collaborative processing. Examples of such applications are interactive video conferencing systems, video-on-demand systems, computer-aided collaborative design and multimedia electronic shopping systems. It is anticipated that the proliferation of multimedia applications development will continue to gain momentum and acceptance just as graphical user interface has replaced traditional command prompt interface. Consequently, there is an increasing need for the development of a software engineering approach to facilitate the modelling and design of a potentially complex multimedia system (Gibbs.1995).

This paper investigates the use of a concurrent, object-based modelling design technique for developing multimedia systems. In particular, the paper describes the use of the PARSE (Parallel Software Engineering) software engineering methodology (Gorton.1995). Issues relating to the modelling of distributed multimedia systems are described in section 2. Section 3 gives a brief overview of the PARSE notations, with an emphasis on tthe features of PARSE that supports distributed multimedia system designs. Section 4 and 5 presents a case study on the design of a video-on-demand (VOD) multimedia application using PARSE, and a mapping to the WIN32 Application Programming Interface (API) is given in section 6.

2 MODELLING DISTRIBUTED MULTIMEDIA SYSTEMS

The development of software for distributed multimedia applications presents several challenging requirements. These range from the need to transmit and synchronise multiple media types, through to support for distributed interactive processing with real-time performance parameters. The development process is further complicated by the need of applications to support heterogeneous hardware platforms, as well as different device drivers and operating systems. Specifically, a software engineering methodology for distributed multimedia systems should support the following key design aspects:

Synchronisation: In multimedia applications, the synchronisation between different types of medium is important. For example, it is crucial to control and synchronise the broadcast of video and audio components of a video segment. In addition, as different components may operate at different speeds, control messages are often sent to several processes in order to synchronise the total system activity. Any design methodology for distributed multimedia software must cater for the explicit design of synchronisation mechanisms.

Dynamic Process and Communication Path Creation: The client-server paradigm can often be found in distributed multimedia applications, where the server process objects spawn extra helper processes to carry out the work as requested by clients. The software design method should be able to capture features such as the dynamic creation and deletion of processes and communication paths.

Strong Modularity: Modularity is essential in distributed multimedia systems design. Encapsulated software objects representing multimedia system components reduce the development effort, shielding the programmer from the complex details of programming and operating media hardware. Essentially, the object model contains all the necessary functions for the operation of the media device, making direct programming unnecessary (Friesen.1995).

This also promotes portability of code, as objects can be reimplemented for different platforms while maintaining a constant functional interface (see Figure 1).

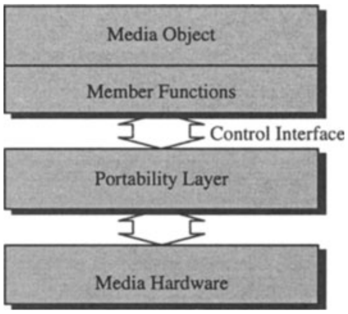


Figure 1 Media Object Interface.

3 OVERVIEW OF PARSE

PARSE (PARallel Software Engineering) is an object-based software engineering methodology that facilitates the design of reliable and reusable parallel and distributed systems. Software design in PARSE is centered around a process graph notation. The notation allows the partitioning and synchronisation of the software to be expressed in a graphical manner. Designs represented as process graphs are simple and concise, and can be progressively refined to capture all the structural and dynamic properties of a design. The three basic features of PARSE process graphs are: explicit classification of process objects using a small set of system supplied general classes; interaction between process objects is done via message passing on typed communication paths; and the unique feature of path constructors, used to specify relationships between process object communication paths. The basic design components of PARSE are shown in Figure 2, and for a full description of the individual components, please refer to (Gorton.1995).

The basic PARSE process graph notation cannot handle the design of distributed systems which incorporate the dynamic creation and deletion of processes and of communication paths. Hence, the dynamic reconfiguration of systems cannot easily be captured in the design. In a typical multimedia application, the provision of these facilities is vitally important. The Extended-PARSE (Ext-PARSE) (Liu.1996) process graph notation (Figure 2) is designed to supplement the basic PARSE process graph notation for this purpose.

Process objects may be created and deleted dynamically. New process objects may enter/exit the system at run time. This should not affect the execution of other processes. Function servers and control process objects may create and delete dynamic process objects by *invoking* create and or delete signals. They are shown via the *twisted arrow* notation. There are two rules of usage: the process object at the *invoked* end of a creation/deletion arrow must be of *dynamic* type; and the process object at the *invoking* end of a creation/deletion arrow must be an *active* process object (this means data server is excluded). There are also three ways that dynamic process objects may exit from the system: *assassination*, where an active process object kills a dynamic process object; *suicide*, where a thread terminates its own execution; and *aging*, which is the default termination mode. The created thread dies from aging when it completes its work. This occurs naturally, hence the term *aging*. The notation caters for all three possible ways of termination, and leaves the design decision for the software engineer.

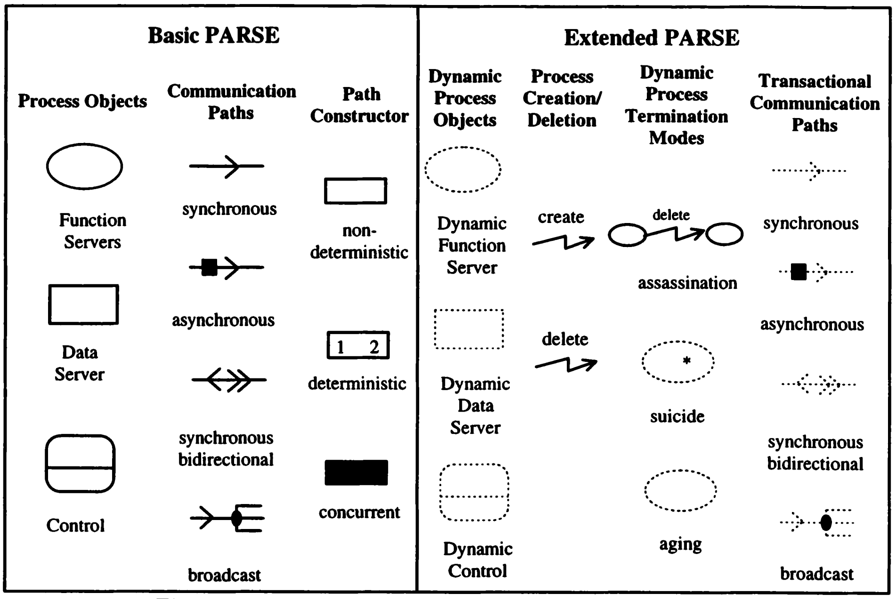


Figure 2 Summary of the PARSE Process Graph Notation.

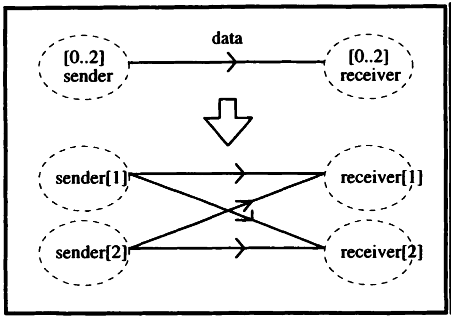
Communication paths going into and or coming out from dynamic process objects are dynamic in nature. These communication paths are set up when the associated process objects are created, and are destroyed when process objects terminate.

Dynamic process objects are often replicated. Each replicated instance has the same internal behaviour. However, not all instances of the process are created simultaneously: different instances may be created and terminated at different times. The series of numbers enclosed by square brackets [0..n] denotes the range of the number of instances of the object that may be present in the system at any one time. The symbol 'n' may be replaced by a constant integer, or by default, is the maximum number of thread instances a process may have as defined by the underlying system.

In Figure 3a, there is (at any time) a maximum of one communication path between the two dynamic process objects. However, this is not always the case. By default, a communication path going into or coming out from a dynamic process object is replicated if there are multiple instances of the process.

By default, all associated processes are fully connected by communication paths. Specific path restriction notations can also be used to override default behaviours (see Figure 3b).

Transactional Communication Path: Software designers can explicitly show that the communication between two processes is of a transactional type by using dotted arcs (see Figure 2 Extended PARSE). In many software systems, such as database applications, *transactional communication paths* are often used. These are different to the ordinary communication paths in the sense that they are set up only when they are needed to transfer messages. As soon as the transfer is complete, the path is no longer valid. Hence, the life span of a communication path is not dependent on the life spans of the processes using it, but is dictated by the activity of transferring the message.



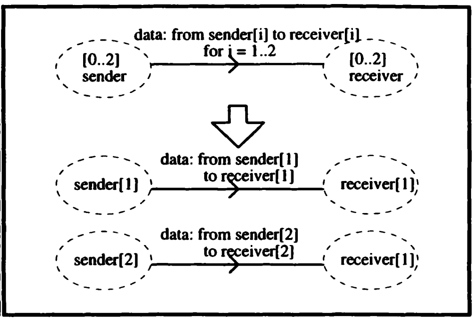


Figure 3a **Figure 3b**

Figure 3a Replication Of Dynamic Communication Path.
Figure 3b Path Restriction.

The period of validity of the communication path depends on the associated process internals. This can be explicitly defined using the behavioural specification language [Gorton95].

PARSE uses hierarchical decomposition to handle large designs. Further, in typical client-server systems, it is often desirable for processes to spawn new helper processes to service multiple clients' requests. *Multithreaded objects* provide abstraction for the low level process creation/deletion activities. The default structure (expressed within the roundangle) is as shown in Figure 4.

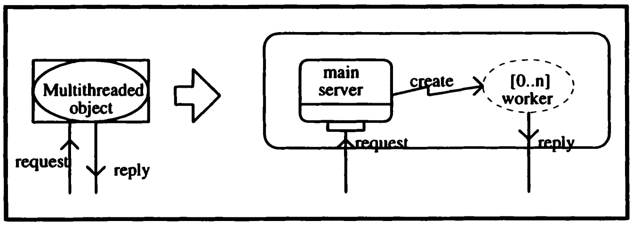


Figure 4 Default Behaviour Of A Multithreaded Object.

A designer may override this default structure and or behaviour by providing the decomposition of the multithreaded object, thus the multithreaded object simply provides an abstraction at the higher level of design, for the low level parallelism.

4 CASE STUDY: VIDEO-ON-DEMAND

Previous work with PARSE has focused on the design and development of closely-coupled parallel systems (Gorton.1994). In this paper, we wish to illustrate the use of PARSE to design loose-coupled distributed systems such as multimedia applications.

This section describes a case study on the use of PARSE for the design of a distributed video-on-demand (VOD) multimedia application[1]. In addition to providing VCR-like functions for controlled playback, the application is designed to support interactive non-linear access to video footage. Compressed videos are manually segmented, and a descriptor file is created for each significant segment of the file. Each descriptor consists of several control parameters and pointers to the start and end of the associated video segment. This control information is loaded into the main memory at start-up time, so as to facilitate high-speed retrieval of the requested video segments. The video server is designed to support multiple concurrent connections. For each new connection requested by a remote client, the parent server process creates a child process to handle the newly requested service.

4.1 PARSE Process Graph Design For VOD

The top-level PARSE design diagram shows the client-server structure of the video-on-demand application (see Figure 5).

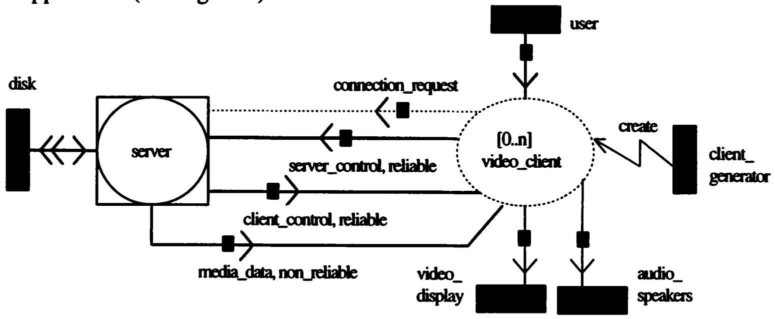


Figure 5 Top Level Diagram.

The video client is modelled as a dynamic process object. The notation *[0..n]* denotes that there may be *0* or more active clients at any one time. Here, *n* is unspecified, and hence only limited by system resources.

A *video_client* is created dynamically by the external *client_generator*[2], which represents the external interface of the system (here, a *video_client* is created in response to the user starting up the application). Upon creation, the client software sends the initial *connection_request* message via the 'transactional' communication path. This prompts the server to initiate a connection for the client (for more details, the internal structure of the multithreaded server is shown in Figure 6). For each connection service requested by a newly created client, a set of connection paths are established for the exchange of control and media information between the distributed client and server processes. The *server_control* connection path is used by the

[1] The precise application details are commercial in confidence.

[2] The solid bar represent an 'external entity' in the PARSE notation (Gorton.1995).

client process to transfer control messages to the server for controlled playback and process synchronisation. Similarly, the *client_control* connection path is used by the server process to provide feedback control information to the client. These control paths are specified as asynchronous in nature, and of type *reliable*, which specifies the protocol used on these paths. Logical protocol definitions are specified textually using a simple protocol definition language: they are omitted from here due to space restrictions. In contrast, the *media_data* path is declared as *unreliable*, as this can survive information loss, but requires low delay and jitter control to support continuous media stream playout at the client.

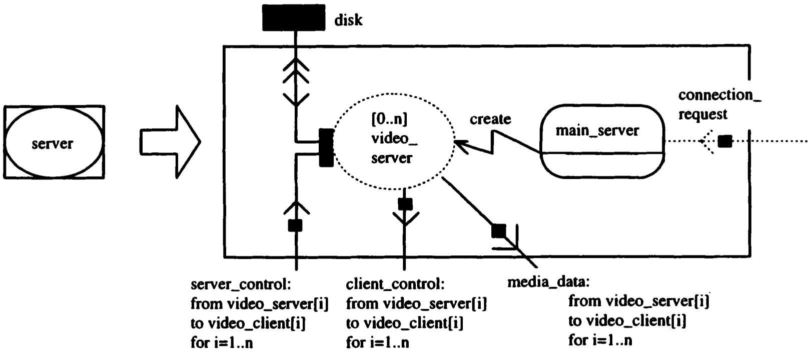

disk
server
[0..n] video_ server
create
main_server
connection_ request
server_control: from video_server[i] to video_client[i] for i=1..n
client_control: from video_server[i] to video_client[i] for i=1..n
media_data: from video_server[i] to video_client[i] for i=1..n

Figure 6 Behaviour of Multithreaded 'Server' Object.

The communications characteristics specified in PARSE can be mapped directly to the appropriate communication Application Programming Interface (API) supported by the underlying network. For example, in an ATM network, the use of reliable communication path specified in PARSE can be mapped to a communication API based on TCP/IP using AAL5 adaptation layer (Boudec.1992). Similarly, the use of an unreliable communication path with specified delay and jitter parameters for continuous media communication can be mapped directly to an API that supports real-time traffic such as AAL 1 and AAL2 (Boudec.1992).

The behaviour of the 'multithreaded server' process is decomposed as shown in Figure 6.

The *video_server* process object is created dynamically in order to service the video clients' requests. This activity is co-ordinated by the *main_server* control process object.

Hence, for each instance of the client process, there is a copy of video server to service the client's request. That is, for each *client[i]*, a *video_server[i]* is created, for *i = 0..n*. The *path restriction* feature (Gorton.1995) in PARSE has been used to specify that the communication paths: *server_control*, *client_control*, and *media_data* only exist between corresponding video server and client.

The video server process is further decomposed in Figure 7.

The server control unit receives control data from the client via the *server_control* communication path. The control information is parsed and the appropriate control actions are sent to *MJPEG_file_server* for processing. Based on the control actions received from the control unit, *MJPEG_file_server* retrieves the appropriate video segment from the disk via the bi-directional communication path, and sends the *media_data* to the client across the network. In addition, *MJPEG_file_server* is responsible for flow control synchronisation with the client to ensure that the client's media buffer is within the lower and upper buffer mark. This is accomplished by having the client process periodically feedback the buffer information and the rate in which the media is being played (via *client_control*). This feedback information is processed by the file server and the appropriate control actions are invoked to ensure correct synchronisation. For example, *if server_control_unit* detects a progressive increase in the client's media buffer such that the buffer's upper threshold has been reached, a control message is sent immediately to the *MJPEG_file_server* to reduce the media transmission rate.

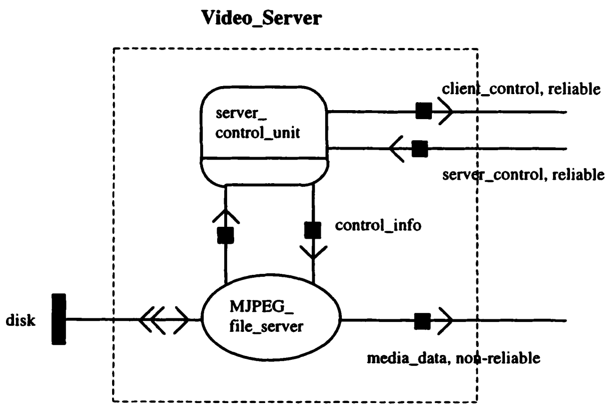


Figure 7 Video Server.

Figure 8 shows the decomposed design of the *video_client* process.

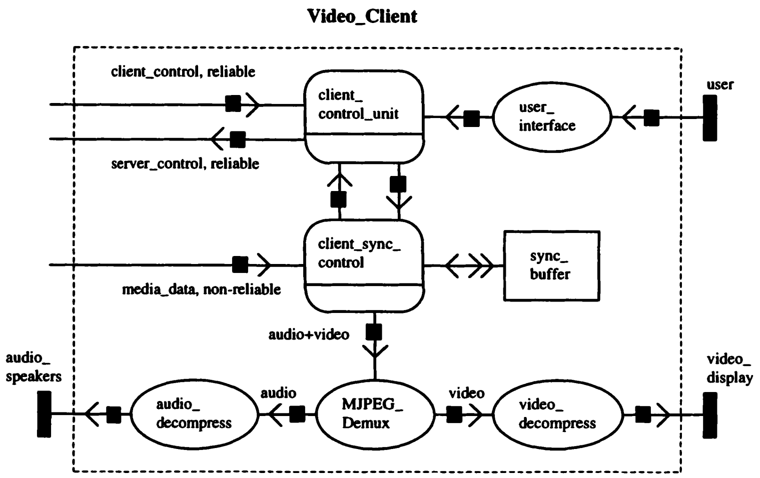


Figure 8 Video Client.

The user interface function is responsible for the processing of user's input. Based on the command received from the user interface function, *client_control_unit* is responsible for invoking the appropriate actions for the local and remote server processes. For example, upon receiving a pause command from the user, the control unit issues a control message to the local client synchronisation unit to stop processing. Subsequently, a control message is also issued to inform the remote server to temporarily cease sending any further media data so as to prevent buffer overflow at the client. The client synchronisation control unit (*client_sync_control*) is responsible for synchronous playout of the continuous media data. For each block of interleaved audio and video frame, based on the frame's playout rate, the synchronisation control unit is responsible to ensure the synchronous playout of the continuous media. In addition, the unit is also responsible for the synchronisation between the client and server to ensure that proper flow control is enforced. The interleaved audio and video data received from the synchronisation control unit is demultiplexed by the *MJPEG_Demux* function unit. The demultiplexed video and audio data unit are sent to the respective decompression functions for media decompression and subsequent playout.

5 SPECIFYING PROCESS OBJECT BEHAVIOUR USING BSL

The dynamic behaviour of the primitive (lowest-level) process objects is specified using a behavioural specification language (BSL). This language contains constructs for describing sequential program structures and includes sequences, iterations, selections, and guarded selections. Primitive send and receive operations for various kinds of communication path types are also included. In addition, dynamic creation, deletion of processes and communication paths can be specified. For a full description of the syntax and usage of BSL, see (Gorton.1995).

The specification of all primitive process objects is beyond the scope of this paper. We will however demonstrate the use of BSL by providing partial descriptions of the behaviour of two primitive process objects taken from Figure 6 and Figure 8. BSL descriptions should fully describe the relative order of inputs and outputs for a given process object: internal processing of inputs can be left unspecified, at least initially. This gives a skeleton specification of the process object's interactions which is sufficient to simulate and verify the system's behaviour (Russo.1995).

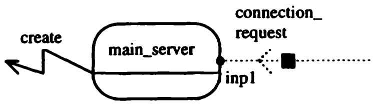


Figure 9 'Main_Server' From Multithreaded 'Server' Object.

```
PROCESS MAIN_SERVER
      SEQUENCE
         WHILE TRUE
            -- main_server sets up transactional communication path
            setup (connection_request)

            -- take input from asynchronous path Connection_request
            -- i is the instance of the client object, -1 denotes
      -- untimed comm.
            receive (inp1, i, -1, connection_request)

            --create helper process after receiving
--Connection_request
```

```
            create (video_server)
         ENDWHILE
      ENDSEQUENCE
END PROCESS
```

The main_server process object has a simple BSL description. It simply waits for a client process to connect via the *connection_request* dynamic path, and creates a *video_server* process object to service further requests. *MJPEG_Demux* specifies that for each input message received on the *audio+video* path, a corresponding output message is sent to both output paths. These completed behavioural descriptions can then be translated into programming languages using APIs supported by the underlying platform.

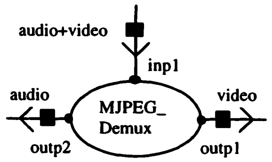


Figure 10 'MJPEG_Demux' From 'Video_Client'.

```
PROCESS MJPEG_DEMUX
      SEQUENCE
         WHILE TRUE
            receive (inp1, i, -1, audio+video)
            -- separates the two types of medium
            as-send (outp1, i, -1, video)
            as-send (outp2, i, -1, audio)
         ENDWHILE
      END SEQUENCE
END PROCESS
```

6 IMPLEMENTATION SUPPORT

Windows NT has been chosen to be the implementation platform. The Application Programming Interface (API) Win32 provides a rich set of function calls that enables the development of distributed multimedia systems. Table 1 gives a summary of Win32 API calls that supports the implementation of any Ext-PARSE designs.
Note: actual Win32 and socket function calls are in *italic*.

7 CONCLUSION AND FURTHER WORK

A number of software design techniques for distributed systems exist. For example: Booch (Booch.1991), CODARTS (Gomaa.1993), HOOD (Robinson.1992), MOOD (Lee.1994), and PROOF (Yau.1994). However, they do not exhibit the ability to easily capture the dynamic, distributed system structures and complex synchronisation requirement frequently occurring in multimedia applications.

This paper has demonstrated the suitability of PARSE for the design of distributed multimedia systems. The complete PARSE process graph notation enables the dynamic system structure to be captured precisely, and succinctly. Multithreaded process objects, and the hierarchical process object structuring promotes a high level of design abstraction. In typical client-server systems, there are often multiple instances of the server object created for servicing multiple clients' requests. The multithreaded process object in PARSE allows the designer to specify this dynamic interaction between client and server easily. The information captured in the textual annotations, such as path restriction and dynamic process object replication ranges, further aids the eventual implementation.

Ext-PARSE process graph feature	**Win NT/Win32 API Equivalent**
Dynamic process objects	Processes, threads
• create dynamic process objects	• *CreateThread, CreateProcess, CreateRemoteThread*
• delete dynamic process objects: - aging - assassination - suicide	• thread termination: - return from function - *TerminateThread* - *ExitThread*
• data server objects	• thread: Thread-Local Storage, shared memory: synchronisation objects
Communication Paths	NamedPipes
• creation	• *CreateNamedPipe* (an instance of a named pipe is always deleted when the last handle to the instance of the named pipe is closed)
• synchronous	• blocking send, blocking receive, overlapped mode not enabled, *pipe-specific mode = PIPE WAIT*
• asynchronous	• non-blocking send, blocking receive, *pipe open mode = overlapped, pipe-specific mode = PIPE WAIT*
• bi-directional synchronous	• blocking send, blocking receive, *pipe open mode = PIPE ACCESS DUPLEX*
• broadcast	• multiple instances of named pipes, or mailslots
Network Communication Paths	Socket facility provided in winsock.h
• creation	• creating new socket, *socket, bind*
• synchronous	• reliable path with blocking send, blocking receive, *socket_type = SOCK_STREAM*
• asynchronous	• non-blocking send, blocking receive, for reliable path *socket_type = SOCK_STREAM,* non-reliable path *socket_type = SOCK_DGRAM*
• bi-directional	• sockets are inherently bi-directional
• broadcast	• multiple instances of sockets. In some networks, eg. ATM, efficient implementation can be achieved using network multicast facility.
Path Constructors	Input selection methods
• unspecified (not used in primitive processes)	• no need to consider this in Win NT
• concurrent	• independent threads carrying out work separately
• non-deterministic	• NamedPipes non-deterministic by default
• deterministic	• *WaitNamedPipe*

Table 1 Mapping between Ext-PARSE and Win32

The PARSE graphical design method would seem intuitive to use and easily comprehensible. In this project, the lead designer produced a PARSE design for the system in a matter of days, with no previous PARSE exposure. We intend to quantitively explore this issue further through additional, controlled case studies.

The verification of designs is the issue that needs to be considered. Currently, static PARSE designs can be easily translated into Petri Nets (Gorton.1994), and subsequent design verification can be carried out automatically. However, this technique can not be used with PARSE designs with dynamic properties, since any reachability analysis would lead to an infinite state system. It is anticipated that an alternative verification technique would be employed. The works of (Birkinshaw.1995) and (Milner.1980) may offer possible solutions.

References

Birkinshaw, C.I. and Croll, P.R. (1995) Modelling the Client-Server Behaviour of Parallel Real-Time Systems Using Petri Nets, *Proc. 28th Ann. Hawaii Int'l Conf. System Sciences, Parallel Software Engineering Minitrack, Vol.2: Software Technology*, IEEE Computer Society Press, Calif., 339-48.

Booch, G. (1991) Object-oriented Design With Applications. Benjamin/Cummings.

Boudec, J.Y.L.(1992) The ATM: A Tutorial, *Computer Networks and ISDN Systems*, Vol.24, 279-309.

Friesen, J.A., Yang, C.L. and Cline, R.E. (1995) DAVE: A Plug-and-Play Model for Distributed Multimedia Application Development, *IEEE Parallel and Distributed Technology*, Vol.3, No.2, Summer, 22-8.

Furht, B. (1994) Multimedia Systems: An Overview, *IEEE Multimedia*, Vol.1, No.1, Spring, 37-50.

Gibbs, S.J. (1995) Multimedia Programming: objects, environments, and frameworks, ACM Press.

Gomaa, H.(1993) Software Design Methods for Concurrent and Real-Time Systems, Addison-Wesley.

Gorton, I., Chan, T.S. and Jelly, I.E. (1994) Engineering high quality parallel software using PARSE, in *Lecture Notes in Computer Science 854*, Proceedings of CONPAR-VAPP 94, Linz, Austria, September, 381-92, Springer-Verlag.

Gorton, I., Gray, J.P. and Jelly, I.E. (1995) Object-based Modelling of Parallel Programs, *IEEE Parallel and Distributed Technology*, Vol.3, No.2, Summer, 52-63.

Jadav, D. Choudhary, A. (1995) Designing and Implementing High-Performance Media-on-Demand Servers, *IEEE Parallel and Distributed Technology*, Vol.3, No.2, Summer, 29-39.

Lee, P.J., Chen, D.J. and Chung, C.G. (1994) An Object-oriented Modelling Approach To System Design, *Information and Software Technology*, Vol.36, No.11, 683-94.

Liu, A. and Gorton, I. (1996) Modelling Dynamic Distributed System Structures in PARSE, to appear in *4th Euromicro Workshop on Parallel and Distributed Processing*, Braga, Portugal, January.

Microsoft Corp. (1991) Microsoft Windows Multimedia Programmer's Workbook.

Milner, R. (1980) A Calculus of Communicating Systems, in *Lecture Notes in Computer Science Volume 92*, Springer-Verlag.

Minzer, S.E. (1989) Broadband ISDN and Asynchronous Transfer Mode (ATM), *IEEE Communications Magazine*, September, 17-24.

Robinson, P.J. (1992) HOOD, Prentice-Hall, 1992.

Russo, S., Savy, C., Jelly, I.E. and Collingwood, P.C. (1995) Petri Net Modelling of PARSE Designs, *Joint Technical Report*, Computing Research Centre, Sheffield Hallam University/ Departmento di Informatica e Sistemistica, University of Naples.

Taylor, H., Chin, D. and Knight, S. (1995) The Magic Video-on-Demand Server and Real-Time Simulation System, *IEEE Parallel and Distributed Technology*, Vol.3, No.2, Summer, 40-51.

Wallace, G.K. (1991) The JPEG Still Picture Compression Standard, *Communications of ACM*, Vol.34, No.4, April, 30-44.

Yau, S.S. and Bae, D-H. (1994) Object-oriented and Functional Software Design for Distributed Real-time Systems, *Computer Communications*, Vol.17, No.10, October, 691-8.

6

Hypersequential Programming

— A Novel Paradigm for Concurrent Programming —

Naoshi Uchihira, Shinichi Honiden, and Toshibumi Seki
Systems & Software Engineering Laboratory, Toshiba Corporation
70, Yanagi-cho, Saiwai-ku, Kawasaki 210, Japan
Telephone: +81-44-548-5690, *Fax:*+81-44-520-5855, *E-mail:* uchi@ssel.toshiba.co.jp

Abstract

This paper proposes *hypersequential programming* which is a novel paradigm for concurrent programming to ease the difficulty of concurrent programming and make the concurrent program highly reliable. The difficulty of concurrent programming is due mainly to its nondeterminism; nondeterminism being the purpose of the concurrent program. We classify nondeterminism into 3 types: *intended*, *harmful*, and *persistent* nondeterminism. In traditional concurrent programming, a programmer first designs and implements programs so as to maximize concurrency, which may include the 3 types of nondeterminism. He then tries to detect harmful nondeterministic behaviors by testing and debugs them. However, it is actually very hard to remove all harmful nondeterministic behavior. On the contrary, in hypersequential programming the concurrent program is first serialized to remove all types of nondeterminism, and then the programmer tests and debugs it as a sequential program. Finally, it is parallelized by restoring only intended and persistent nondeterminism. With hypersequential programming, a highly-reliable concurrent program can be developed because the injection of harmful nondeterminism is precluded. This paper shows the generic concept and a simple embodiment of hypersequential programming.

Keywords

concurrent programming, nondeterminism, serialization, parallelization, dependence analysis, highly-reliable program, hypersequential programming, default sequential principle

1 INTRODUCTION

Generally speaking, we find it more difficult to develop concurrent programs than we do sequential programs. In testing and debugging of concurrent programs, the combination of data and timing variations causes an explosive increase of behavior complexity and often produces unexpected (probably harmful) behaviors. Moreover, concurrent programs do not always have reproducible behavior (McDowell and Helmbold, 1989).

These difficulties are due mainly to their capricious (i.e., nondeterministic) behaviors. Nondeterminism can be classified into the following 3 types.

- **Intended nondeterministic behavior:** Nondeterministic behavior which the programmer intends to implement.
- **Harmful nondeterministic behavior:** Nondeterministic behavior which the programmer does not intend to implement and does not expect.

- **Persistent nondeterministic behavior:** Nondeterministic behavior which is race-free, that is, has no effect on the results.

In conventional concurrent programming, the programmer tries to detect and remove harmful nondeterministic behavior in testing and debugging. However, it is actually very hard to remove all harmful behavior by testing.

This paper proposes a novel programming paradigm which is the reverse of the conventional programming and is applicable to actual concurrent programs. In hypersequential programming, all types of nondeterminism are removed at first by serialization, whereas only harmful nondeterminism is removed in the case of conventional programming. Then the programmer tests and debugs the serialized program in the conventional way. Finally, intended and persistent nondeterminism is restored by parallelization. While serialization and parallelization should be computer-aided, testing and debugging are basically done in the conventional way. Hypersequential programming makes concurrent programming as easy as sequential programming because testing and debugging are done for serialized programs, and thus high productivity and high reliability can be achieved.

The remainder of the paper is organized as follows. Section 2 explains three types of nondeterminism by example, and illustrates a concept of hypersequential programming. Then, we introduce a generic procedure of hypersequential programming and fundamental techniques for it in Section 3. Section 4 shows a concrete embodiment of hypersequential programming using a simple example. Finally, Section 5 mentions related works, and conclusions are presented in Section 6.

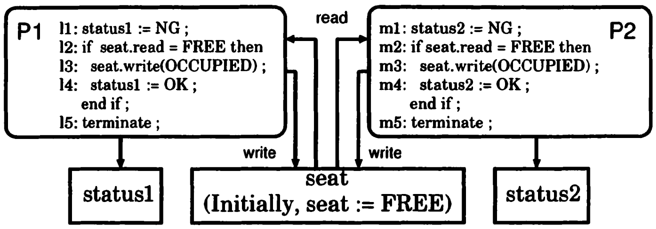
2 NEW PARADIGM FOR CONCURRENT PROGRAMMING

This section explains the concept of hypersequential programming. First, we classify nondeterminism of concurrent programs into 3 types in detail. Then, we illustrate how hypersequential programming differs from conventional concurrent programming with regard to manipulation of nondeterminism.

2.1 Concurrency and Nondeterminism

Even when a concurrent program runs with the same input, its behavior can be different. This is nondeterminism. Nondeterministic behaviors can be classified into three types: *intended*, *harmful*, *persistent* (Uchihira and Honiden 1995). We explain them using a simple example shown in Figure 1. The example is a simple Ada-like concurrent program "*Seat Booking*", where two processes read/write a shared memory "*seat*" to reserve one seat. This program has the following nondeterministic behavior, which can be classified into 3 types.

- $\theta_1 = l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_1 \rightarrow m_2 \rightarrow m_5$
  Result: $status_1 = OK, seat = OCCUPIED, status_2 = NG.$
- $\theta_2 = m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow m_4 \rightarrow m_5 \rightarrow l_1 \rightarrow l_2 \rightarrow l_5$
  Result: $status_1 = NG, seat = OCCUPIED, status_2 = OK.$
- $\theta_3 = l_1 \rightarrow m_1 \rightarrow l_2 \rightarrow m_2 \rightarrow l_3 \rightarrow m_3 \rightarrow l_4 \rightarrow m_4 \rightarrow l_5 \rightarrow m_5$
  Result: $status_1 = OK, seat = OCCUPIED, status_2 = OK.$
- $\theta_4 = \mathbf{l_1} \rightarrow \mathbf{m_1} \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_2 \rightarrow m_5$
  Result: $status_1 = OK, seat = OCCUPIED, status_2 = NG.$
- $\theta_5 = \mathbf{m_1} \rightarrow \mathbf{l_1} \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_2 \rightarrow m_5$
  Result: $status_1 = OK, seat = OCCUPIED, status_2 = NG.$
- ....



```
P1  l1: status1 := NG ;
    l2: if seat.read = FREE then
    l3:   seat.write(OCCUPIED) ;
    l4:   status1 := OK ;
        end if ;
    l5: terminate ;
```

```
m1: status2 := NG ;                P2
m2: if seat.read = FREE then
m3:   seat.write(OCCUPIED) ;
m4:   status2 := OK ;
    end if ;
m5: terminate ;
```

Figure 1 An example of a concurrent program

Intended nondeterminism The nondeterministic behaviors θ_1 and θ_2 derive different results: P_1 can book the seat ($status_1 = OK$) in θ_1, but cannot ($status_1 = NG$) in θ_2. However both are correct (intended behaviors).

Harmful nondeterminism The nondeterministic behavior θ_3 derives an incorrect result (double booking). Thus, this program has harmful nondeterminism.

Persistent nondeterminism The nondeterministic behaviors θ_4 and θ_5 have the same result because l_1 (write in $status_1$) and m_1 (write in $status_2$) are actions independent of each other. We call such a situation *persistent.*

Note that all intended nondeterministic behavior must be implemented, while persistent nondeterministic behavior is permitted but not necessarily implemented. Harmful nondeterministic behavior is wrongly injected when the programmer is implementing intended and persistent behaviors.

2.2 Paradigm Shift

Old Paradigm

In our observation of conventional concurrent program development, a programmer first tries to design and implement processes so as to maximize concurrency, which may include the 3 types of nondeterminism ($\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, ...$). He then tries to detect harmful nondeterministic behaviors (θ_3) in testing and debugs them by partially serializing the critical sections which interfere with each other using synchronization mechanisms (e.g., semaphore, rendezvous). Bugs due to harmful nondeterministic behavior often account for a considerable part of all timing bugs. In conventional concurrent programming, it is very difficult and requires heavy load to remove all harmful nondeterministic behavior for large-scale programs. Moreover, there is no assurance that all harmful behaviors are removed, and some bugs still remain more often than not.

This situation is illustrated in Figure 2. The dense tree denotes a concurrent program which has a lot of bugs (i.e., harmful behaviors). In conventional concurrent programming, bugs are removed one by one during testing and debugging. However, since the tree is dense, it is very hard to find all bugs, and some bugs remain.

New Paradigm

A novel programming paradigm, called *hypersequential programming,* is the reverse of the old paradigm. In a nutshell, the hypersequential programming consists of serialization and parallelization. In hypersequential programming the programmer first serializes the concurrent program to remove all types of nondeterminism, and then tests and debugs it as a sequential program. Finally, he parallelizes it by introducing only intended and persistent nondetermin-

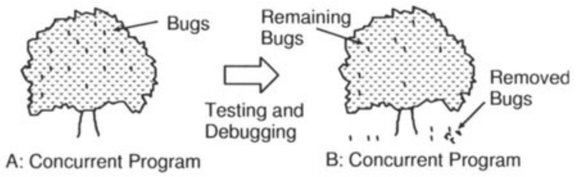


Figure 2 Conventional Concurrent Programming (Old Paradigm)

ism. We claim that hypersequential programming promotes a paradigm shift in concurrent programming.

In the case of the example (Figure 1), the program can be serialized by introducing a process priority ($P_1 > P_2$). The serialized program has only one behavior θ_1, which can be tested as easy as for a sequential program. Then, another intended behavior θ_2 is appended to the program. Finally, persistent nondeterministic behaviors $\theta_4, \theta_5, \ldots$ are restored by automatic parallelization.

This new paradigm is illustrated in Figure 3. After serializing the concurrent program represented by a dense tree A, only the trunk of the tree remains. This bare tree B illustrates the program stripped of all types of nondeterministic behaviors. It is easy to remove bugs from the bare tree as compared with the original dense tree, which implies that a lot of bugs (i.e., harmful nondeterministic behaviors) are removed together with serialization and further, the serialized program can be tested and debugged as easily as a sequential one for remaining bugs. However, this bare tree cannot fulfill the original requirements. It should be restored to its original dense condition. First, intended nondeterministic behaviors are explicitly introduced by the programmer. This program (we call it hypersequential program C) can fulfill the functional requirements but lacks the parallel speedup; thus the tree looks ill-formed. Then, persistent nondeterministic behaviors are restored. This introduction can be automatically done by using parallelization techniques. The programmer finally gets a new concurrent program D which may be slightly different from the original dense tree A but has no bugs.

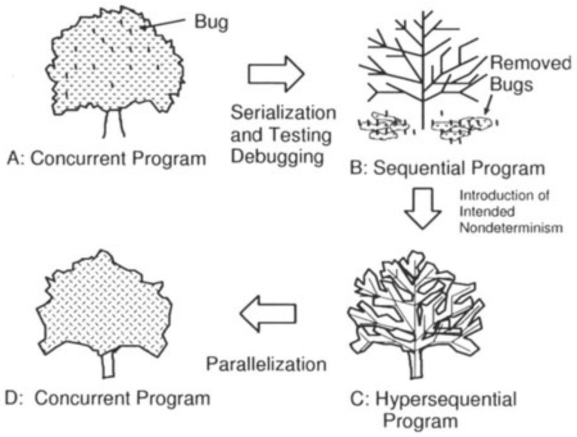

Bug
A: Concurrent Program
Serialization and Testing Debugging
Removed Bugs
B: Sequential Program
Introduction of Intended Nondeterminism
C: Hypersequential Program
Parallelization
D: Concurrent Program

Figure 3 Hypersequential programming (New Paradigm)

3 HYPERSEQUENTIAL PROGRAMMING

Hypersequential programming consists of the following five steps.

Step 1: Modeling and Coding Model the target system as it is, using concurrency and nondeterminism naturally. Then code it with a concurrent programming language.

Step 2: Serialization (Projection) Serialize the concurrent program to remove all types of nondeterminism. The generated program can be viewed as a projection of the concurrent program onto a sequential program, which we call a *hypersequential program* *.

Step 3: Testing and Debugging
Test and debug the hypersequential program. Since the program is serialized, testing and debugging are as easy as with a sequential one.

Step 4: Introduction of Intended Nondeterminism
Introduce intended nondeterministic behaviors into the hypersequential program. After each introduction, the program should be tested again for added behaviors.

Step 5: Parallelization (Restoration)
Parallelize the program automatically by permitting persistent nondeterministic behaviors. This parallelization signifies the restoration of concurrency.

In a nutshell, a concurrent program is first projected into a sequential dimension which facilitates the programmer's job, and then the program is restored again into concurrent dimension. Each step is explained in detail in the following sections.

3.1 Serialization

The target concurrent program is serialized in the meta-level control and converted into a hypersequential program. There are several methods of the meta-level control for serialization as follows.

- **Serialization based on global priorities:** By introducing global priorities among processes, methods, or program sections, a concurrent program becomes deterministic for the same inputs. Note that the global priority control assumes a virtual single CPU environment; it is impractical to implement global control (i.e., global ordering of events) actually for parallel and distributed environments.
- **Serialization based on event histories:** The execution of a concurrent program can be recorded as an event history. A concurrent program can be executed deterministically under the control of the event history.
- **Serialization based on scenarios:** Instead of event histories, the programmer can specify execution order among program sections by giving a scenario.

Serialization techniques have been developed for debugging concurrent programs in order to solve the reproducibility problem (Bernstein and So, 1991). However, serialization information is used only for testing and debugging in these works, while serialization information (i.e., *default execution order* explained in 3.2) is also used for parallelization in hypersequential programming.

*A hypersequential program is a program which is serialized at the meta-control level and preserves the topology of the original concurrent program. After introducing intended nondeterminism in Step 4, the hypersequential program may have some partial concurrency.

3.2 Hypersequential Program

A hypersequential program is composed of *section information*, *section graph*, *program dependence graph*, and *serialization information.*

- **section information:** The original program (source code) is divided into program sections. A program section is a segment of a parallel program that is coded to be executed by one process. A program section may be a *basic block* which has no branching and synchronization except its start and end, and can be executed deterministically and be automatically extracted. The programmer can also specify a program section explicitly using some landmark point (we call it *section point*) as delimiter. In this case, program section may consist of several basic blocks and include some branching and synchronization, where process switching is allowed only at section points and prohibited during execution of the section.
- **section graph:** A section graph represents a topological control structure of the original program, where nodes correspond to program sections.
- **program dependence graph:** A program dependence graph (Ferrante, et al, 1987) represents both control dependence and data dependence between program sections in a single graph.
- **serialization information:** Serialization information specifies execution order among program sections. This execution order consists of two types of order: *a priori* execution order defined in the original program and *a posteriori* execution order given by serialization. The latter order is called *default execution order* which should be preserved as far as parallelization steps do not remove it explicitly. This execution order may be total order just after serialization, but will change to partial order after parallelization steps.

3.3 Introduction of Intended Nondeterminism

Introduction of intended nondeterminism is a new and important step which does not exist in the conventional programming. There are several methods to introduce intended nondeterminism.

- **Removing default execution orders:** The programmer can remove some default execution orders of the hypersequential program which generate intended nondeterministic behaviors. In other words, total ordering of sections is relaxed into partial ordering. Figure 4 illustrates a simple example of several removing default execution orders. Note that an a priori execution order cannot be removed.
- **Enumerating essential scenarios:** The programmer enumerates all essential scenarios, which correspond to test cases in conventional programming. The essential scenarios should be representative of all intended behaviors. A set of intended behaviors can be restored from these scenarios by parallelization.

3.4 Parallelization

Parallelization techniques have been extensively studied for compiler optimization for supercomputers (Zima and Chapman, 1990). In hypersequential programming, these parallelization techniques are used to detect persistent nondeterministic behaviors and restore them. This parallelization should preserve the original semantics, that is, it must not generate nondeterministic behavior which can produce results different from the original one.

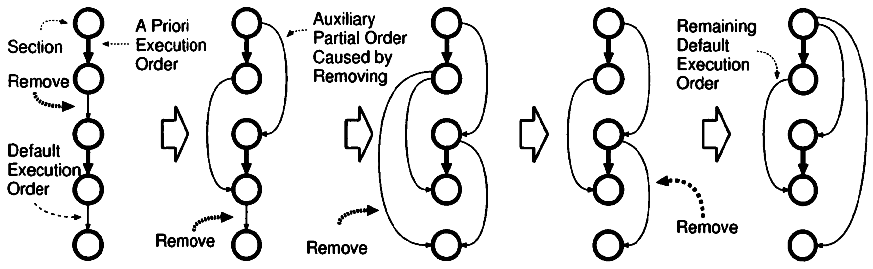


Figure 4 Removing Default Execution Orders

Concretely, a hypersequential program is parallelized as follows. First, precedence constraints between sections are automatically extracted from the dependence graph and the serialization information. If two sections have no precedence constraints, they can be parallelized. Then, default execution order between sections in the serialization information can be removed if they have no precedence constraints. Finally, a concurrent program is generated, where the remaining default execution order in the serialization information is preserved by inserting synchronization codes to implement it.[†] The resulting program involves only intended and persistent nondeterministic behaviors.

We want to emphasize that some of initial default execution orders by serialization still remain as far as they are not explicitly removed by introduction of intended nondeterminism or parallelization. These remaining default orders preclude the injection of harmful nondeterminism. We call it the *default sequential principle.*

4 SIMPLE EMBODIMENT

Since hypersequential programming is a rather conceptual paradigm, there are a variety of concrete procedures based on the concept. This section shows a simple embodiment of hypersequential programming using Petri nets.

4.1 Simple Example

We now explain hypersequential programming by a simple example. The target concurrent program consists of two processes P_1, P_2 on different processors, and two shared memories M_a and M_b. Although this program has no branch and loop, it can demonstrate essential features of hypersequential programming.

Step 1-1: Modeling and coding a target program

The target concurrent program $P = P_1 || P_2$ is described as in Figure 5.

Step 2-1: Setting program sections

In this case, assume that each instruction forms one section. To simplify the description, an instruction code is itself used as a section ID.

[†]The remaining default execution order can be also implemented by run-time control instead of inserting synchronization codes into the program itself.

```
P1:                                            P2:
begin                                          begin
init1 ;    /* Initialize Memory Ma     */      init2 ;    /* Initialize Memory Mb      */
read1 ;    /* Read Data from Memory Mb */      read2 ;    /* Read Data from Memory Ma  */
write1 ;   /* Write Data in Memory Ma  */      write2 ;   /* Write Data in Memory Mb   */
end                                            end
```

Be careful! M_a is accessed by $init_1, read_2, write_1$, and M_b is accessed by $init_2, read_1, write_2$.

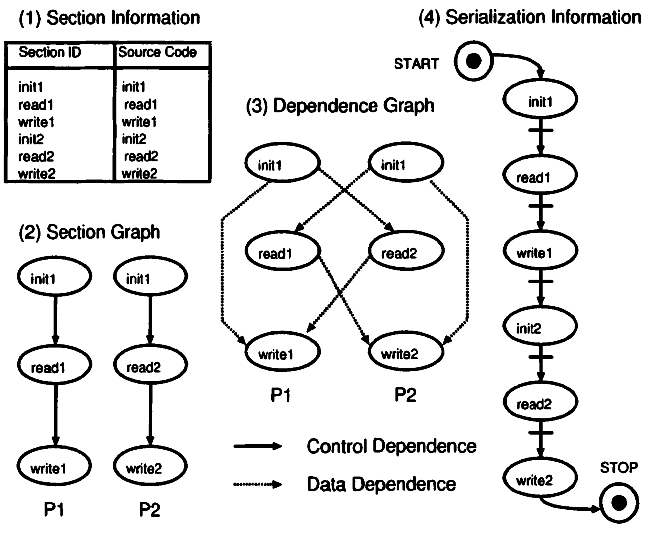
Figure 5 Target Concurrent Program $P = P_1 \| P_2$ (Source Code)

Step 2-2: Serializing the concurrent program

The concurrent program P is serialized by introducing a process priority $P_1 > P_2$ which means P_1 is executed with the higher priority than P_2. In this case, the execution order of sections is represented as follows.

$$init_1 \rightarrow read_1 \rightarrow write_1 \rightarrow init_2 \rightarrow read_2 \rightarrow write_2$$

After serialization, the hypersequential program HSP is represented as shown in Figure 6, which consists of *section information*, *section graph*, *program dependence graph* and *serialization information*. The serialization information represents the above execution order of sections using Petri nets.



(1) Section Information

Section ID	Source Code
init1	init1
read1	read1
write1	write1
init2	init2
read2	read2
write2	write2

Figure 6 Hypersequential Program HSP

Step 3: Testing and debugging hypersequential program

The hypersequential program HSP is executed and tested by the programmer. If bugs are detected, the source code of sections is corrected. If a bug is related to the program structure, the original concurrent program is modified and serialized again. In this case, a bug such that $read_1$ accesses M_b before M_b is initialized is detected. Then the programmer debugs it by inserting synchronization commands (*send*, *wait*). The modified concurrent program is shown in Figure 7, where $read_1$ accesses M_b after M_b is initialized. This program should be serialized and tested again.

```
P1:                                          P2:
begin                                        begin
init1 ;    /* Initialize Memory Ma     */    init2 ;    /* Initialize Memory Mb     */
wait(1) ;  /* Synchronization          */    send(1) ;  /* Synchronization          */
read1 ;    /* Read Data from Memory Mb */    read2 ;    /* Read Data from Memory Ma */
write1 ;   /* Write Data in Memory Ma  */    write2 ;   /* Write Data in Memory Mb  */
end                                          end
```

Figure 7 Modified Concurrent Program (Source Code)

Step 4-1: Introduction of intended nondeterminism

The Petri net representing the serialization information is displayed. The programmer introduces intended nondeterminism explicitly by removing default execution orders represented by arrows of Petri nets. In this case, a default execution order between $write_1$ and $read_2$ is removed (Figure 8(a)).

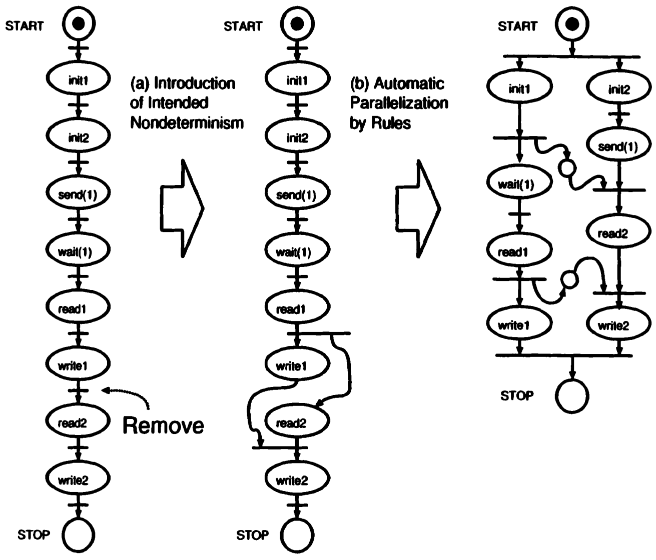

START
init1
init2
send(1)
wait(1)
read1
write1
read2
write2
STOP
Remove
(a) Introduction of Intended Nondeterminism
(b) Automatic Parallelization by Rules

Figure 8 Introduction of Intended Nondeterminism

Step 4-2: Testing and debugging hypersequential program

The hypersequential program into which nondeterminism is introduced is executed and tested again. In this case, the program has two nondeterministic behaviors.

- $init_1 \rightarrow init_2 \rightarrow send(1) \rightarrow wait(1) \rightarrow read_1 \rightarrow$ **write₁** $\rightarrow$ **read₂** $\rightarrow write_2$, or
- $init_1 \rightarrow init_2 \rightarrow send(1) \rightarrow wait(1) \rightarrow read_1 \rightarrow$ **read₂** $\rightarrow$ **write₁** $\rightarrow write_2$.

If bugs are detected, the program should be corrected. In this case, these two behaviors are intended ones and no bugs are detected.

Step 5-1: Automatic parallelization

Precedence constraints among program sections can be automatically derived by analyzing the serialization information and the program dependence graph, especially data dependence among sections. Default execution orders with some precedence constraints should be preserved in parallelization because the execution result may change depending on the execution order of sections having data dependence. In this case, there are the following precedence constraints: $init_1 \rightarrow read_2$, $init_2 \rightarrow read_1$, $read_1 \rightarrow write_2$. For example, a value read by the process P_1 in $read_1$ is influenced by whether $write_2$ of the process P_2 occurs before or after $read_1$. Therefore, the default execution order $read_1 \rightarrow write_2$ in the serialization information should be kept in parallelization. Sections having no precedence constraints can be executed concurrently, and can be parallelized. For example, since two sections $read_1$ and $read_2$ have no precedence constraint, they can be parallelized, i.e., the default execution order $read_1 \rightarrow read_2$ is removed.

This parallelization can be done automatically by applying rules shown in Figure 9 to transform the target Petri net. Figure 8(b) shows a parallelized Petri net after rule application.

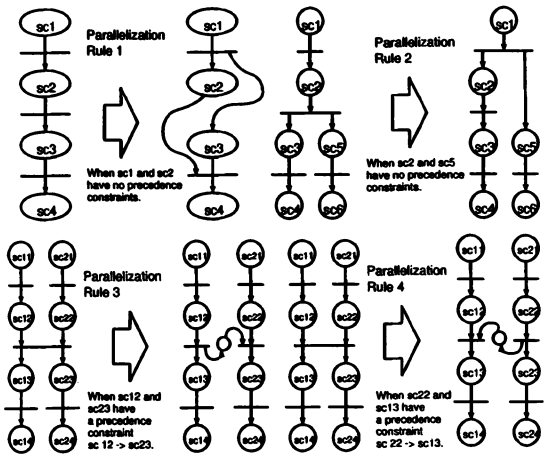

Parallelization Rule 1
When sc1 and sc2 have no precedence constraints.
Parallelization Rule 2
When sc2 and sc5 have no precedence constraints.
Parallelization Rule 3
When sc12 and sc23 have a precedence constraint sc 12 -> sc23.
Parallelization Rule 4
When sc22 and sc13 have a precedence constraint sc 22 -> sc13.
sc1
sc2
sc3
sc4
sc5
sc6
sc11
sc12
sc13
sc14
sc21
sc22
sc23
sc24

Figure 9 Rules for Parallelization

Step 5-2: Generation of Concurrent Program

The source code of the final concurrent program is generated from the hypersequential program after parallelization, where the remaining default execution order should be implemented. In this case, synchronization instructions (*send* and *wait*) are embedded in the source code for realizing the default execution order; $init_1 \rightarrow read_2$ and $read_1 \rightarrow write_2$. The generated concurrent program is shown in Figure 10.

4.2 Treatment of Branches and Loops

Ordinary programs have usually branches and loops. This section considers briefly treatment of branches and loops in hypersequential programming.

```
P1:
begin
init1;      /* Initialize Memory Ma     */
send(2);    /* synchronization          */
wait(1);    /* synchronization          */
read1;      /* Read Data from Memory Mb */
send(3);    /* synchronization          */
write1;     /* Write Data in Memory Ma  */
end
```

```
P2:
begin
init2;      /* Initialize Memory Mb     */
send(1);    /* synchronization          */
wait(2);    /* synchronization          */
read2;      /* Read Data from Memory Ma */
wait(3);    /* synchronization          */
write2;     /* Write Data in Memory Mb  */
end
```

Figure 10 Generated Concurrent Program (Source Code)

Steps concerning serialization, testing and debugging, and introduction of intended nondeterminism are done in the same way. Only the parallelization step requires additional consideration concerning branches and loops. In general, this parallelization has been well investigated and several techniques are available. In Petri-net-based parallelization, we use Petri net folding/unfolding techniques for branching. For example, the *rule*5-8 (Figure 11) shows parallelizing program fragments by folding/unfolding nets. With regard to loops, we basically adopt a hierarchical parallelization approach (Girkar and Polychronopoulos, 1992), in which loops are treated as single hierarchical sections and a program is represented as an acyclic section graph in each hierarchical level. Parallelization is done for each hierarchical level. Furthermore, loop unwinding rules are applied to promote parallelization effectively. The *rule*9 (Figure 11) is one of the loop unwinding rules.

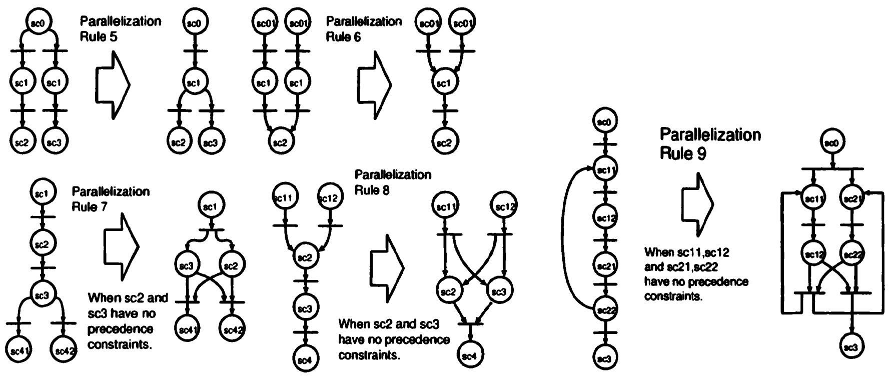

Parallelization Rule 5
Parallelization Rule 6
Parallelization Rule 7
When sc2 and sc3 have no precedence constraints.
Parallelization Rule 8
When sc2 and sc3 have no precedence constraints.
Parallelization Rule 9
When sc11,sc12 and sc21,sc22 have no precedence constraints.
sc0
sc01
sc1
sc2
sc3
sc4
sc41
sc42
sc11
sc12
sc21
sc22

Figure 11 Rules for Parallelization (Branch and Loop)

5 RELATED WORKS

When developing concurrent programs, it is useful to test and debug them after serializing them and deleting their nondeterminism. For example, we usually test and debug concurrent programs in a single CPU environment acting as a pseudo-multi-CPU environment before doing so in an actual multi-CPU environment. Bernstein and So (1991) systematize this debugging know-how. They proposed a debugging method for concurrent programs by stepwise serialization. With regard to parallelization of sequential programs, a great deal of research has been done in the domain of compiler optimizations for parallel computers. Zima and Chapman (1990) summarized these techniques. Recently, automatic extraction of task-level parallelism has been studied by Girkar and Polychronopoulos (1992,1995). However, there are no reports of research into the combination of serialization and parallelization. The reader may ask why not starting with sequential program instead of a concurrent one which is serialized afterward. Answer is that topology of concurrent program is very natural for many target domains. In hypersequential programming, the topology of the original program is preserved at serialization, and it is restored at parallelization.

The concurrency control of database transactions (Bernstein and Goodman, 1981) is related to the techniques of hypersequential programming. The concurrency control is intended to remove harmful nondeterminism and leave intended and persistent nondeterminism as much as possible. However, it is intended only for database transactions, and not for ordinary concurrent programs. Hypersequential programming is aimed at testing and debugging of ordinary concurrent programs which may themselves have synchronization codes.

6 CONCLUDING REMARKS

Hypersequential programming can make concurrent programming as easy as sequential programming, and produce a highly reliable program. This paper shows the basic concept and approach of hypersequential programming and a simple embodiment of the concept. We think this concept is very general and there may be a lot of embodiments, and it can be applied to wide domains of concurrent programming. At the same time, since hypersequential programming is still in its infancy, a lot of techniques need to be developed in order to put hypersequential programming to practical use.

Acknowledgments: The concept of hypersequential programming is the fruit of a pooling of ideas and discussion in a certain research project being carried out by Toshiba corporation. We would like to thank our fellow researchers on the project: Hideaki Shiotani, Akihiko Ohsuga, Satoshi Itoh, Shinsuke Sawashima, Mitsuru Kakimoto, Yasuo Nagai, Yasuyuki Tahara, Katsuhiko Ueki, Keiichi Handa, and Tamiya Ochiai. We are also grateful to Sadakazu Watanabe and Kazuo Matsumura of the Systems & Software Engineering Laboratory, Toshiba Corporation, for their support throughout this work.

REFERENCES

McDowell, C.E. and Helmbold, D.P. (1989) Debugging Concurrent Programs, *ACM Computing Surveys*, Vol.21, No.4.

Uchihira, N. and Honiden, S. (1995) Compositional Adjustment of Concurrent Programs to Satisfy Temporal Logic Constraints in MENDELS ZONE, *28th Hawaii International Conference on System Science (HICSS)*.

Bernstein, D. and So, K. (1991) Debugging Parallel Programs by Serialization, *United States Patent* No. 5048018.

Ferrante, J., Ottenstein, K.J., and Warren, J.D. (1987) The Program Dependence Graph and Its Use in Optimization, *ACM Trans. on Programming Languages and Systems*, Vol.9, No.3.

Girkar, M. and Polychronopoulos, C.D. (1992) Automatic Extraction of Functional Parallelism from Ordinary Programs *IEEE Trans. Parall. Distrib. Syst.*, Vol.3, No.2.

Girkar, M. and Polychronopoulos, C.D. (1995) Extracting Task-Level Parallelism *ACM Trans. Prog. Lang. Syst.*, Vol.17, No.4.

Zima, H. and Chapman, B. (1990) *Supercompilers for Parallel and Vector Computers*, Addison-Wesley.

Bernstein, F.A. and Goodman, N. (1981) Concurrency Control in Distributed Database Systems, *ACM Computing Surveys*, Vol.13, No.2.

7

Efficient composition and automatic initialization of arbitrarily structured PVM programs

J.Y. Cotronis
Dept. of Informatics, University of Athens
TYPA Builds., Panepistimiopolis, 157 71 Athens, GREECE
tel.: +30 1 7291885 fax: +30 1 7219 561 e-mail: cotronis@di.uoa.gr

Abstract

There are significant programming and methodological problems when developing PVM programs, the process communication structure of which does not form trees but arbitrary graphs. We present a design methodology, called Ensemble, and the appropriate PVM techniques and tools for the efficient composition of arbitrarily structured PVM programs. In Ensemble PVM programs are described by annotated Process Communication Graphs (PCGs) and the sequential program components are designed with open communication interfaces. The annotated PCGs are interpreted by a universal PVM program Loader which spawns processes and sets values to their communication interfaces, thus establishing the program communication structure. The program components are reusable without any modification in other PVM programs. Annotated PCGs are produced from PVM program scripts. The methodology may be applied to any message passing environment by developing specific annotations of the PCG, reusable program components and the program loader.

Keywords

PVM, program composition, reusable components, annotated process communication graphs

1 INTRODUCTION

PVM allows for the most general form of MIMD parallel computation, as programs in PVM may possess arbitrary control and dependency structures (Geist et al., 1994). At any point in the execution of a PVM program, the processes in existence may have arbitrary relationships between each other and any process may communicate and/or synchronize with any other. As with all programming environments there are program categories that are well suited to the PVM characteristics, making them easy to implement, and others that are not well suited and are much more difficult to implement. Let us overview PVM's fundamental characteristics and examine their influence on the design and implementation of programs:

1. The underlying architecture of PVM is any host system running UNIX and some special cases for Massively Parallel architectures, which are viewed as virtual machines.
2. Hosts may run the PVM console, which allows the user to interactively start, query, modify the virtual machine. PVM programs may use the complete host system; distinct programs may use the same hosts.
3. A PVM process is a UNIX process running on a host machine. A process is spawned by its parent process. To run a PVM program the user spawns a root process.
4. Processes are identified by unique integer identifiers, called task identifiers (tid), which are generated upon process creation by pvm_spawn. The tid is only known to the spawning process and the spawned process may obtain its father's and its own tid by function calls.
5. Processes may be spawned at specific hosts. If no host is specified PVM chooses where to spawn them. There are also some other spawning options.
6. Process communication and synchronization are of two categories: 1) requiring process tids and possibly some message tag identifiers (tags), such as point to point asynchronous communication (pvm_send, pvm_recv, etc.) and muticast, sending the same value to a list of processes, and 2) requiring group definitions, such as bcast, sending the same value to processes in a group and barriers, where groups of processes synchronize.

Programming applications forming, in general, tree-like process communication dependencies, where each process communicates only with its parent and its children processes, is easy to program in PVM, as for example SPMD and master/slave programs. The parent process spawns its children processes and each child process obtains its parent's tid. However, programming arbitrarily structured programs, in which the dependency structures of processes form arbitrary graphs is not, in general, an easy task in PVM. Establishing graph-like process communication dependencies in PVM requires a substantial programming effort is in two directions:

1. Creating the processes according to the parent-child model.
2. Establishing the full graph communication. Processes have to obtain the tids of the processes with which they need to communicate. They already know their children's tids and they may easily obtain their parent's tid. The programmer has to program processes to obtain the tids of the rest of the processes with which it needs to communicate.

As arbitrary process graph structures are to be established, ad-hoc programming is used which depends on the specific communication dependencies of the program in hand. The explicit programming of the order of creation of processes and of establishing the communication has the following disadvantages:

1. It is an overhead effort, since it is enforced by PVM (parent-child process creation and identification of processes upon their creation) and not by the program specification.
2. It burdens the design and implementation of programs, since the extra coding makes programs more difficult to understand, to debug and to modify.
3. It limits the reusability and scalability of program components, since components involve code which relies on specific dependency structures.

In this paper we present the Ensemble methodology and its techniques and tools for the efficient composition and initialization of arbitrarily structured static PVM programs overcoming the above disadvantages. The Ensemble methodology comprises three facets:

1. The annotated Process Communication Graphs (PCGs). We use general PCGs, as a natural structure, representing the processes as nodes and the communication dependencies between them as arcs. PCGs have been extensively used in modeling (Andrews, 1991), in dynamic analysis and simulation (Pouzet et al., 1994; Schneider and Schaefers, 1993), in mapping techniques (Norman and Thanish, 1993), etc. We annotate nodes and arcs of PCGs with information a PVM program needs for the creation and communication of its processes. We consider the annotated PCGs as interpretable structures specifying the composition of PVM programs. The annotated PCGs are produced from program scripts, but may also be produced by a graphical tool.
2. The reusable program components. Processes in PVM are spawned by loading instantiations of executable program components. We have developed programming structures and principles for program components which permit their reusability as executable library components. Such program components do not assume any specific communication structure in which the processes instantiated from them are involved. They specify a general parametric interface with the type and possibly the number of its communication dependencies and all the actual parameters of the communication procedures, such as pvm_send and pvm_recv, refer to elements of the interface. A reusable program component may have any number of instantiations in the same PVM program, as well as in other PVM programs, each instantiation having its own communication dependencies. When a process is instantiated it should be given appropriate information setting values to its interface. This information is annotating the PCG and is sent by the PVM Loader.
3. The PVM Loader. A universal PVM process which automatically initializes PVM programs by interpreting the annotated PCGs. The PVM Loader visits the nodes of the annotated PCGs and spawns the appropriate processes (instantiations of reusable components) according to the annotation on the nodes. The Loader then sends the process interface information annotating the PCG to the processes it spawned.

The structure of the paper is as follows: in section 2 we present the annotated Process Communication Graphs and their script representation; in section 3 we present the structure and design principles of reusable programs in PVM; in section 4 we present the PVM Loader; in section 5 we demonstrate the methodology by composing PVM programs all consisting of two types of reusable components. In section 6 we present our conclusions and plans for future work.

2 THE ANNOTATED PROCESS COMMUNICATION GRAPHS

Before we define the PCGs and their annotation let us describe a distributed application which we shall use as a demonstrating example.

2.1 A distributed application: Get Maximum

There are processes instantiated from a terminal component which possess a value; all terminal processes or simply terminals need to get the maximum value possessed by any of them. To limit the number of messages the terminals do not broadcast their own value to all others; instead, there are processes instantiated from a relay component to which groups of terminals send their values, for simplicity their tids. The relay processes or simply relays cooperate to find the maximum of the tids, which they then send to their respective groups of terminals.

The terminals have one communication dependency, that with their associated relay, which we call S (Server) type. The relays have two types of communication dependencies, one with their groups of terminals, which we call C (Client) type, and one with the relays, which we call P (Propagation) type. A relay may have any non negative number of C dependencies and P dependencies. The main actions of terminals and relays are:

The actions of a terminal		**The actions of a relay**	
send tid to relay	(to S type)	receive tids from the client terminals	(from C type)
receive maximum tid from relay		find the local maximum LM of tids	
	(from S type)	send LM to all other relays	(to P type)
		receive LMs from all other relays	(from P type)
		find the global maximum GM	
		send GM to its client terminals	(to C type)

We like the program to be easily configurable, that is, to be possible to add or remove any number of terminal and/or relay processes, without any modification of the program components, i.e. the terminal and relay executables..

2.2 The elements of the PCG and their annotation

Processes will be depicted on PCGs by nodes comprised of two concentric circles (Figure 1). On the inner circle the type of dependencies are indicated. The inner circle depicts the general interface type of the program components. The arcs leaving the nodes indicate communication dependencies (of a specific type) with other processes. The points where the arcs cut the outer circle depict the actual interface of processes to other processes. Each point of intersection is called a **port** and is indexed by a unique positive integer within a port type. The arcs of the PCG connect ports of nodes. Under this scheme terminals and relays will be depicted on PCGs as in Figure 1 (a) and (b) respectively.

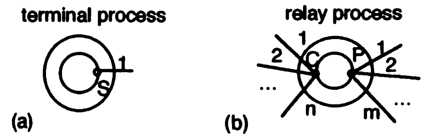


Figure 1 Graphical depiction of terminal and relay processes.

Let us assume, for example, that we have a configuration of eight terminals connected to four relays. The three C type ports of relay R[1] are connected with the S type ports of three terminals T[1], T[2] and T[3]; the two C type ports of relay R[2] are connected with the S type ports of two terminals T[4] and T[5]; the two C type ports of relay R[3] are connected with the S type ports of two terminals T[6] and T[7]; and finally the single C type port of R[4] is connected with the S type port of T[8]. All relays are connected to each other via their P ports. The PCG depicting the process dependencies is shown on Figure 2. The ports are indexed and connected according to the described configuration. As a matter of convenience the nodes are indexed by positive integers. The elements of the PCG described so far specify a general PCG independent of any parallel implementation system.

Arcs on a PCG represent communication dependencies. For a complete communication specification in PVM, request identifiers, called tags, are needed which are used by both sending and receiving processes. The tag identifiers annotate the arcs of the PCG. In Figure 2 the arcs are annotated by unique positive integers, shown in bold.

Nodes may be further annotated by allocation information, if a process is to be spawned on a particular host. Finally, nodes are annotated by the full path name of the executable from which the process it represents will be instantiated. For reasons of simplicity allocations and executable path names are not depicted on Figure 2.

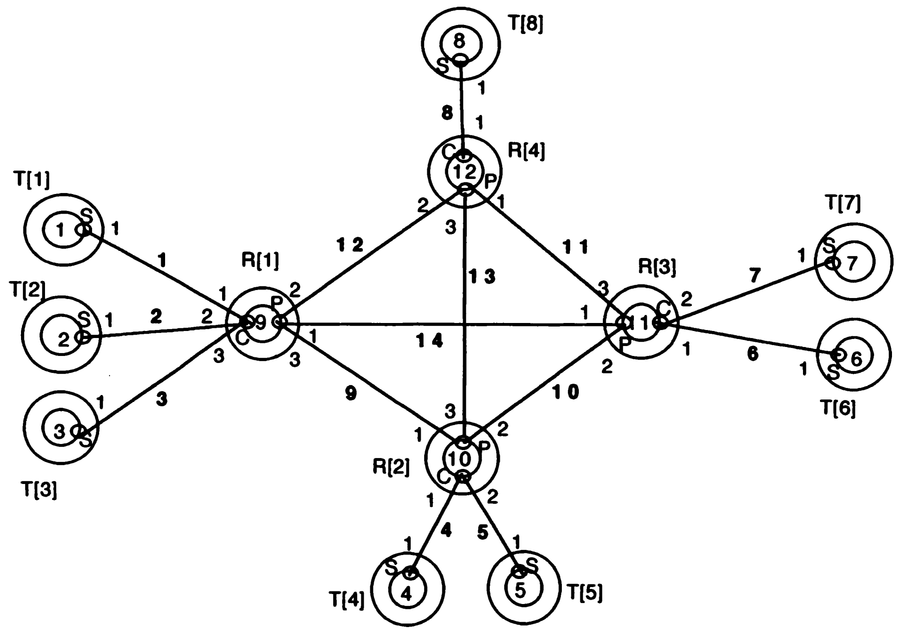


Figure 2 The annotated PCG of the application Get Maximum.

The annotated PCG may be interpreted by the PVM Loader to initiate the program. The annotated PCG may be produced by a graphical tool or by a textual description. We have developed a script language and programs which read a program script and produce the annotated PCG. A program script has three sections: the first describes the general PCG, the second the annotation of the PCG specific to a parallel environment (in this case PVM) and the third the annotation specific to the sequential components.

The script generating the annotated PCG of Figure 2, is presented in two columns in Figure 3. The first section, headed with `PCG`, defines the nodes and the number of ports for each type (e.g. all T nodes have one port of type S); it also defines the connections between the ports. The second section, headed with `Parallel System` defines the specific PCG annotation for the PVM. The compulsory annotation for RequestID, annotating the arcs, is specified; here the default specifies the annotation of the arcs by unique positive integers, but generating algorithms or direct annotations may be defined. Also optional annotation may be specified; here all processes are allocated on specific hosts. The third section, headed with `Sequential System`, annotates the nodes of the PCG with the file locations of the executables of the sequential components from which processes are to be instantiated. From the program scripts annotated PCGs are produced which are interpreted by the PVM Loader initiating the PVM program.

```
                              Application | Get Maximum
```

```
PCG
Components
/* specify for each process the
number of ports of each type*/
 T[1], T[2], T[3], T[4], T[5],
 T[6], T[7], T[8] #ports = S:1;
 R[1]           #ports = C:3, P:3;
 R[2], R[3]     #ports = C:2, P:3;
 R[4]           #ports = C:1, P:3;
Connections
/* Connect process ports */
 T[1].S[1] <-> R[1].C[1];
 T[2].S[1] <-> R[1].C[2];
 T[3].S[1] <-> R[1].C[3];
 T[4].S[1] <-> R[2].C[1];
 T[5].S[1] <-> R[2].C[2];
 T[6].S[1] <-> R[3].C[1];
 T[7].S[1] <-> R[3].C[2];
 T[8].S[1] <-> R[4].C[1];
 R[1].P[1] <-> R[3].P[1];
 R[1].P[2] <-> R[4].P[2];
 R[1].P[3] <-> R[2].P[1];
 R[2].P[2] <-> R[3].P[2];
 R[2].P[3] <-> R[4].P[3];
 R[3].P[3] <-> R[4].P[1];
```

```
Parallel  System
   environment  PVM3;
    PVM3_annotation
    RequestID : default; /* annotate arcs
                by integer request Ids */
    PVM3_allocation
    /* specify the hosts on which
       processes are to be spawned */
    R[1], T[1], T[2], T[3]  at orion;
    R[2], T[4], T[5] at zeus;
    R[3], T[6], T[7] at ismini;
    R[4], T[8] at adonis;

Sequential  System
   Location
  /* full path and name of executables */
    R:
    "/home/users/easy_spawn/bcast/relay";
    T:
  "/home/users/easy_spawn/bcast/terminal";
```

Figure 3 The script of the PVM program for Get Maximum

3 THE DESIGN OF REUSABLE PVM PROGRAM COMPONENTS

Reusability of compiled program components in a message passing environment demands that their process instantiations should be possible to establish the communication dependencies required by parallel programs. As the number of process instantiations and their communication dependencies cannot be fixed, the program components should specify the number and type of communication dependencies in a general way. They should only provide the means for establishing communication between any process instantiated from it with any other processes via an interface.

For establishing a point-to-point communication between PVM processes two values are needed in each process: the tid of the other process and the common tag identifier. Therefore, we define a data structure, called **component port**, having two elements in which (tid, tag) pairs may be stored. A program component may have any number of component ports of the same type, which are organized in an array. Finally, a program component may have many types of component ports. The types of ports form the array **Interface**, the elements of which point to their array of ports. Each port is now identified by its type and its port index within the type.

Upon their creation processes should fix their interface. This involves two actions: the creation of the appropriate number of ports for each type and the setting of value pairs (tid, tag) to the port structures. We permit flexible process interfaces, as program components only fix the type of ports and not the actual number of the ports within types. Each process may have any number of ports of each type. Processes in our methodology are created by the PVM Loader which visits the PCG nodes and spawns processes according to the annotation of the node. The PVM Loader sends the number of ports of each type (depicted on the PCG node) to the process just created. The first action of a process is to receive the number of ports of each type and make the appropriate number of ports of each type in its Interface. This is coded in the `MakePorts` routine.

The value pairs (tid, tag) for each port cannot, in general, be sent at the time of process creation, as a process with which it needs to communicate may have not been spawned yet and its tid would not be known. The (tid, tag) pairs are send to the processes after all of the processes have been spawned, together with the type and index of the port. The processes receive the type, port number, tid and tag and set their Interface accordingly. This activity is coded in the `SetInteface` routine.

In Figure 4 we present the general structure of a reusable program component, which consists of a declaration of the `Interface` structure having N types of dependencies and as actions: a call of `MakePorts`, receiving from the Loader and making the appropriate number of ports of each dependency type; a call of `SetInterface`, receiving from the Loader and setting the values of the ports; and a call of **`RealMain`** which starts the main activity of the component. All PVM reusable components have the same structure; the programmer has only to replace N for the specific number of the types of ports and code the component activity in the **`RealMain`**, in which the parameters of the communication routines **`pvm_send`** and **`pvm_recv`** are expressions of the form **`Interface[S].port[p].tid`** and **`Interface[S].port[p].tag`**, where S is a port type and p is the number of port. By the time a process calls its **`RealMain`** its actual interface would be fixed.

```
void main()
{     InterfaceType Interface[N];
      MakePorts(Interface);
      SetInterface(Interface);
      RealMain(Interface);
}
```

Figure 4 The structure of reusable components in PVM

Having defined the annotated PCGs and the structure of the reusable components, we may describe the final facet of the methodology, the PVM Loader.

4 THE PVM LOADER

The PVM Loader is a universal PVM program by which PVM programs composed according to the methodology are initiated. The PVM Loader takes as input an annotated PCG and visits all its nodes; at each node the Loader spawns an instantiation of the executable file annotating the node. Then sends to the process just created the number of ports of each type and annotates the PCG node with the tid of the process. Having visited all nodes and created all processes, the PVM Loader visits the nodes once more and sends the port interface information (port type, port number, tid, tag) to the processes.

Suppose, in our example (Figure 2), that the PVM Loader visits the node R[2] identified by 10: spawns process R[2], an instantiation of the program component relay and sends to it the number of its ports of each type, as shown in the first column of the following table:

actual values	**explanation**
C, 2	type C has 2 ports
P, 3	type P has 3 ports

The PVM Loader also annotates the node by the process tid, say tid(10). In its second visit to the node the Loader sends to the process identified by tid(10) information to set its interface. The values are shown in the first column of the following table:

actual values	**explanation**
C, 1, tid(4), 4	port 1 of type C is connected to tid(4) with tag 4
C, 2, tid(5), 5	port 2 of type C is connected to tid(5) with tag 5
P, 1, tid(9), 9	port 1 of type P is connected to tid(9) with tag 9
P, 2, tid(11), 10	port 2 of type P is connected to tid(11) with tag 10
P, 3, tid(12), 13	port 3 of type P is connected to tid(12) with tag 13

Running the PVM Loader with the annotated PCG as input we get the following output; the first column is produced by the Loader and the second by the terminal processes:

```
Spawn process 1 (terminal) tid=  c0005 [80004]  The maximum tid is 140005
Spawn process 2 (terminal) tid=  c0006 [100003] The maximum tid is 140005
Spawn process 3 (terminal) tid=  c0007 [80005]  The maximum tid is 140005
Spawn process 4 (terminal) tid= 140004 [140004] The maximum tid is 140005
Spawn process 5 (terminal) tid= 140005 [c0005]  The maximum tid is 140005
Spawn process 6 (terminal) tid=  80004 [c0006]  The maximum tid is 140005
Spawn process 7 (terminal) tid=  80005 [c0007]  The maximum tid is 140005
Spawn process 8 (terminal) tid= 100003 [140005] The maximum tid is 140005
Spawn process 9 (relay)  tid= c0008
Spawn process 10 (relay) tid= 140006
Spawn process 11 (relay) tid= 80006
Spawn process 12 (relay) tid= 100004
```

As the twelve processes, eight terminal and four relay are spawned, the PVM Loader prints their tids; the terminal processes print the global maximum of their tids. All terminal processes print the same maximum of #140005 which was the tid of process 5.

For a PVM program to behave correctly, the nodes on the PCG and the actual program components must be compatible, that is, they should specify the former virtually and the latter actually the same number of types of ports. Furthermore, the connections between ports should be of compatible type, that is connected components agree on the type of messages they exchange and their management. The present version of the PVM Loader does not check the compatibility of the connections. We are currently investigating formal methods for describing and testing component compatibility, which will be integrated in the PVM Loader.

The script language is flexible and permits the rapid composition of PVM programs. It is straight forward to edit scripts to scale a program, by adding and connecting new components, to change the allocation of processes to hosts, change the topology of the components, etc., without modifying the program components.

5 VARIATIONS ON THE GET MAXIMUM PROGRAM

The specification for the Get Maximum program in section 2 did not specify any particular topology by which the relay processes should be connected. In our solution of section 2 we had adopted a topology in which all relay processes are connected with each other. We may achieve the same program functionality by adopting different relay topologies. We shall present two variations, one in which relay processes form a star topology and a second in which they form a tree topology. For these variations we will modify the scripts and not the components.

5.1 Get Maximum by star topology

In this solution we use an extra relay process to which the old four relay processes will be connected. The four relay processes have now only one P (propagation) port, through which they send the maximum value received from their terminals. The new relay process, let us call it central, has four ports of type C (clients). The P type ports of the four relay processes are connected to the C type ports of the central process! Let us note, that the C and the P ports of the relay processes are compatible, as only one value is sent and one value received through them. The PCG for this configuration is depicted in Figure 5:

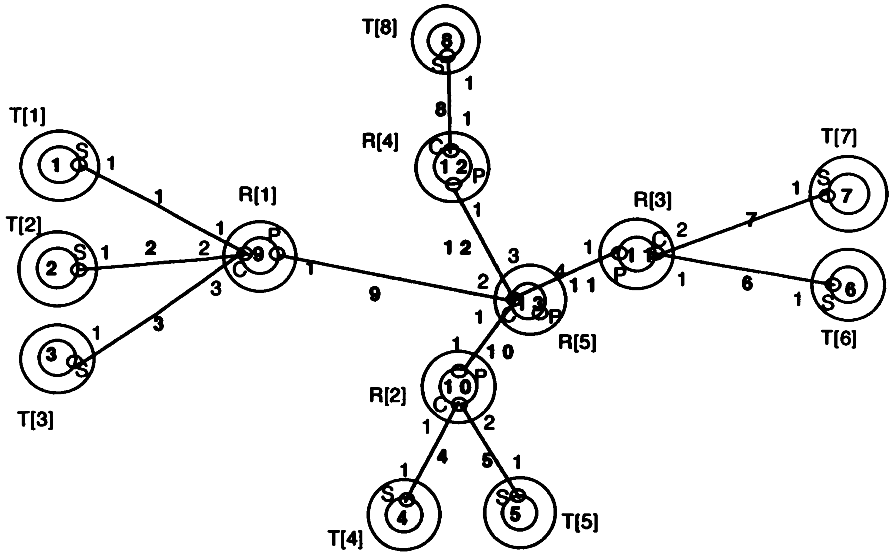


Figure 5 The PCG of Get Maximum by Star Topology.

Let us describe the behavior of the program in such configuration. The four relay processes, as before, select the maximum of the tids of their clients but now propagate it through their single port of type P to the central relay process. The central relay process receives tids from its C ports and selects their maximum. There are no P ports to send the maximum. It then sends its maximum to its C ports. What actually sends is the global maximum, as it is the maximum of all maxima. On its C ports there are the four relay processes. Each receives the global maximum, but, according to the algorithm, they know that it is only the maximum of the central relay process. They compare it with their own maximum, select the value they have received and send it to their client ports. For this solution no changes were made to the terminal or to the relay program components, but only to the script. The executables of the terminal and relay program component were reused. The new program script was produced rapidly by modifying the program script of the version of section 2. From the script the annotated PCG was produced, which was given as input to the PVM Loader. The PCG part of the modified script and the final output of the program are in Figure 6:

```
Get-Maximum-Star
PCG
Components
 T[1], T[2], T[3], T[4],T[5],
 T[6], T[7], T[8]#ports = S:1;
 R[1]        #ports = C:3, P:1;
 R[2], R[3]  #ports = C:2, P:1;
 R[4]        #ports = C:1, P:1;
 R[5]        #ports = C:4, P:0;
Connections
   T[1].S[1] <-> R[1].P[1];
   T[2].S[1] <-> R[1].P[2];
   T[3].S[1] <-> R[1].P[3];
   T[4].S[1] <-> R[2].P[1];
   T[5].S[1] <-> R[2].P[2];
   T[6].S[1] <-> R[3].P[1];
   T[7].S[1] <-> R[3].P[2];
   T[8].S[1] <-> R[4].P[1];
   R[1].P[1] <-> R[5].S[1];
   R[2].P[1] <-> R[5].S[2];
   R[3].P[1] <-> R[5].S[3];
   R[4].P[1] <-> R[5].S[4];
```

```
Program output
Spawn process 1 (terminal) tid= c000e
Spawn process 2 (terminal) tid= c000f
Spawn process 3 (terminal) tid= c0010
Spawn process 4 (terminal) tid= 14000b
Spawn process 5 (terminal) tid= 14000c
Spawn process 6 (terminal) tid= 8000b
Spawn process 7 (terminal) tid= 8000c
Spawn process 8 (terminal) tid= 100008
Spawn process 9 (relay)  tid= c0011
Spawn process 10 (relay) tid= 14000d
Spawn process 11 (relay) tid= 8000d
Spawn process 12 (relay) tid= 100009
Spawn process 13 (relay) tid= 4000a
[100008] The maximum tid is 14000c
[14000b] The maximum tid is 14000c
[14000c] The maximum tid is 14000c
[c000e]  The maximum tid is 14000c
[8000b]  The maximum tid is 14000c
[c000f]  The maximum tid is 14000c
[c0010]  The maximum tid is 14000c
[8000c]  The maximum tid is 14000c
```

Figure 6 The PCG part of the script of the Get Maximum by Star Topology and the output.

5.2 Get Maximum by tree topology

In this variation we maintain the relationship of the eight terminals to the four relay processes having, as in the star solution, one P port. The P ports of R[1] and R[2] are connected with the C ports of R[5] and the P ports R[3] and R[4] are connected with the C ports of R[6]. Both R[5] and R[6] have two C ports and one P port; their P ports are connected to the two C ports of R[7], which does not have any P ports. The process structure is a tree of height 3: the terminal processes as leafs; R[1], R[2], R[3] and R[4] at level two; R[5] and R[6] at level one; and R[7] as the root. At each level, the relay processes receive the values from their clients, select the maximum and propagate it to the next level up. The root selects the maximum and sends it to its client processes. The relay processes below the root do the same until the maximum reaches the terminal processes. The script and the output are shown in Figure 7.

We have demonstrated the flexibility of the methodology by producing non trivial solutions for a program specification using the same reusable components. The program components were reused within the same PVM programs, as well as in other PVM programs. The only changes required were in the program scripts. Although the script language is still under development, it has been successfully used to compose and execute programs from designs very rapidly.

Get-Maximum-Tree

```
PCG
Components
 T[1], T[2], T[3], T[4],T[5],
 T[6], T[7], T[8]    #ports = S:1;
 R[1]               #ports = C:3,P:1;
 R[2], R[3], R[5], R[6]
                    #ports = C:2,P:1;
 R[4]               #ports = C:1,P:1;
 R[7]               #ports = C:2,P:0;
Connections
   T[1].S[1] <-> R[1].C[1];
   T[2].S[1] <-> R[1].C[2];
   T[3].S[1] <-> R[1].C[3];
   T[4].S[1] <-> R[2].C[1];
   T[5].S[1] <-> R[2].C[2];
   T[6].S[1] <-> R[3].C[1];
   T[7].S[1] <-> R[3].C[2];
   T[8].S[1] <-> R[4].C[1];
   R[1].P[1] <-> R[5].C[1];
   R[2].P[1] <-> R[5].C[2];
   R[3].P[1] <-> R[6].C[1];
   R[4].P[1] <-> R[6].C[2];
   R[7].C[1] <-> R[5].P[1];
   R[7].C[2] <-> R[6].P[1];
```

Program output

```
Spawn process 1 (terminal) tid= c0016
Spawn process 2 (terminal) tid= c0017
Spawn process 3 (terminal) tid= c0018
Spawn process 4 (terminal) tid= 140011
Spawn process 5 (terminal) tid= 140012
Spawn process 6 (terminal) tid= 80011
Spawn process 7 (terminal) tid= 80012
Spawn process 8 (terminal) tid= 10000c
Spawn process 9 (relay)  tid= c0019
Spawn process 10 (relay) tid= 140013
Spawn process 11 (relay) tid= 80013
Spawn process 12 (relay) tid= 10000d
Spawn process 13 (relay) tid= 40010
Spawn process 14 (relay) tid= 40011
Spawn process 15 (relay) tid= 40012

[80011]  The maximum tid is 140012
[10000c] The maximum tid is 140012
[80012]  The maximum tid is 140012
[140011] The maximum tid is 140012
[c0016]  The maximum tid is 140012
[140012] The maximum tid is 140012
[c0018]  The maximum tid is 140012
[c0017]  The maximum tid is 140012
```

Figure 7 The PCG part of the script of the Get Maximum by Tree Topology and the output.

6 CONCLUSIONS

We have presented a design methodology, called Ensemble, by which we overcome the problems of composing arbitrarily structured static PVM programs. In the Ensemble methodology parallel PVM programs are virtually specified by annotated PCGs which are interpreted by one universal PVM Loader, spawning the PVM processes and establishing their communication dependencies. We produce PCGs from a script language. Although, the language is still under development it was shown to be flexible and permitted the rapid composition of PVM programs. It is straight forward to edit the script to scale a program, by adding and connecting new components, to change the allocation of processes to hosts, to change the topology, etc. We have proposed simple programming structures and principles for designing reusable PVM program components as library components. Program components are easy to write, as the main actions of program components are wrapped within fixed code segments. The programmer is not concerned with writing code for achieving a process topology.

We demonstrated the flexibility of the methodology, by composing various solutions to the Get Maximum problem. Having constructed the program components for the first solution we used them to compose and execute new PVM programs. This approach is related to the composition of object oriented applications by using objects and scripts (Nierstratz et al., 1991), as it encourages a component oriented approach to application development. We shall pursue this aspect in future work.

The Ensemble methodology is not concerned with the efficiency of program execution. It supports the efficient composition and initialization of applications. The methodology affects the efficiency of the program execution only marginally; before the processes begin their main actions they have to call the `MakePorts` and `SetInterface` routines.

The Ensemble methodology may be applied to other message passing parallel environments by developing specific techniques and tools. We have applied it to the Massively Parallel architecture of PARSYTEC GC3/512 running the Parix environment (Cotronis, 1995). The Parix environment imposes altogether different constraints to programs than PVM. Parix requires different PCG annotation techniques, its own construction of reusable program components and its own Loader. We shall compare implementations of the methodology under PVM, Parix and other environments in a future report. We shall also investigate the portability of parallel programs developed with this methodology. Let us finally comment, that the script language and the structure of the reusable components are such that it seems possible to port programs by editing the annotation parts of scripts and by making new reusable components in the target environment by just changing the "wrapping code" of the `RealMain` procedure in the components.

7 REFERENCES

Andrews, G.R. (1991) Paradigms for Process Interaction in Distributed Programs, *ACM Computing Surveys*, Vol. 23, No.1, March 91.

Cotronis, J.Y. (1995) A Methodology for Initiating Arbitrary Structured Programs in Parix by Interpreting Graphs, in Proceedings of ZEUS 95 (ed. P. Fritzon and L. Finmo) IOS Press.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. (1994) PVM 3 User's guide and Reference Manual, ORNL/TM-12187, May 1994.

Nierstratz, O., Tsichritzis, D., de Mey V. and Stadelmann, M. (1991) Objects + Scripts = Applications, in Proceedings of Esprit 1991 Conference, Kluwer Academic Publishers.

Norman, M.G. and Thanisch, P. (1993) Mapping in Multicomputers. *ACM Computing Surveys*, Vol. 25, No.3.

Pouzet, P., Paris, J. and Jorrand, V. (1994) Parallel Application Design: The Simulation Approach with HASTE, in Proceedings of. HPCN, Munich, Vol II.

Scheidler, C and Schaefers, L. (1993) TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications, in Proceedings of PARLE Conf., Munich.

8 BIOGRAPHY

Dr. J.Y.Cotronis obtained his Ph.D. in Computer Science in 1982 from the Computing Laboratory, University of Newcastle-upon-Tyne, where he also worked as a Research Associate in projects in the area of parallelism. He has been involved in a number of R&D projects in industry and academia. He is an Assistant Professor and his current research interests are on methodologies and supporting tools for composing and porting parallel applications.

Arcadia: A platform for the study of dynamic scheduling of communicating processes

Bernon C., Bétourné C. and Sayah A.
Institut de Recherche en Informatique de Toulouse (IRIT)
Université Paul Sabatier
118 Route de Narbonne - 31062 Toulouse Cedex - France
Tel. (33) 61.55.83.43 Fax (33) 61.55.68.47
E-mail: bernon, betourne, sayah @irit.fr

Abstract
We present a strategy for dynamically schedule communicating processes of a parallel application onto a loosely coupled distributed system. As we must manage two criteria to schedule processes (the workload of the differents sites and the cost of the communication between processes), the algorithm proposed uses two types of agents (system agents and application agents). A system agent manages the workload of its site, an application agent reduces the IPC costs within its application. Our two types of agents cooperate and negociate to make a trade-off between the two criteria. Therefore, we describe the different cooperations needed via a two-layer model. An implementation of this strategy is also described and experimental results analysed. The results obtained justify certain of our design choices and show that the improvement provided by our scheduling algorithm is satisfactory.

Keywords
Dynamic scheduling algorithm, communicating processes, agents, co-operation.

1 INTRODUCTION

The main goal of this paper is to study how an application can run transparently over a distributed network of workstations using dynamic process scheduling. The study is based on several premises:

- the concept of a loosely coupled distributed system, i.e. a system of interconnected workstations communicating over a network, is becoming more and more widespread;
- in such environments, certain machines are overloaded while others remain idle (Krueger,91);
- increasing numbers of system designers are producing parallel applications, but they rarely take full advantage of the resources provided by the underlying distributed system, opting instead for pseudo-parallel execution.

Extensive research has been done on the strategies that can be employed for application process scheduling (Bryant, 81; Ferrari,87; Bernard,93; Ju,95). Our work on the behaviour of a multi-agent system developed at IRIT has led us to identify the following application features:

- an application is made up of a large number of entities executing in parallel;
- these entities have long execution times;
- they communicate by message passing, in an unpredictable fashion;
- the application behaves in a nondeterministic manner.

The second goal of our study was to develop a platform to apply the scheduling strategy adopted, with a view to evaluating the strategy and thus providing input for the design of scheduling algorithms.

We shall now look at the broad outline of this algorithm, the model chosen and the policies adopted to implement it. How the test platform was implemented and the experimental results obtained will be discussed later.

2 ALGORITHM SPECIFICATIONS

Two types of scheduling algorithm exist: those that schedule processes statically before execution (Billionnet,89) and those that schedule them dynamically while an application is running (Bernard,93).

We opted for a dynamic scheduling algorithm for the following reasons:

- static scheduling is less flexible than dynamic scheduling;
- the way processes communicate and how much interaction will take place cannot be predicted;
- it is hard to gauge how an application will behave, and therefore to apply predetermined static schemes.

Further, as the processes created by the applications studied have relatively long execution times, we could also look at ways of moving processes during execution, so a suitable process migration mechanism is supposed to already exist (Nutall,94).

The aim of a dynamic scheduling algorithm is to answer the question 'when do we move a process, which process do we move, and where do we move it ?'. Three policies are usually implemented in response to these issues (Zhou,88):

- the information policy, which determines the type of information required to enable scheduling decisions to be made, and how that information is gathered;
- the transfer policy, which determines whether processes need to be moved, and if so which process should be moved;
- the location policy, which determines which processor should be allocated to the process selected above.

We shall now consider the main features of these three policies with respect to the scheduling algorithm adopted.

2.1 Information policy specification

Information policies usually rely on processor characteristics (system configuration, workload, available memory, etc.) as a basis for scheduling decisions. The processor workload index most often used is the number of processes ready to execute on a CPU at a given moment (Ferrari,87; Kunz,91). We shall adopt a similar index.

Where processes are communicating, we need to consider their characteristics (resources requested, estimated execution time, etc.) and how they interact (relationships, frequency and volume of communications, etc.). Neither of these factors is known in advance. First, by looking at the intensity and volume of intercommunication between processes, we can determine how they interact.

The information policy must then determine to which system processors information applies (to all or just some processors). This information may be gathered on request (Zhou,88), periodically (Litzkow,88), or in response to process state transitions. We opted for a scheme based on state transitions, in which information may be gathered in a centralized or decentralized manner (Douglis,91). We chose the latter solution, given the weak points of centralized management (bottlenecks, lack of fault tolerance, etc.). Information can therefore be communicated between pairs of processors (Bryant,81), distributed to all other system processors (Disted in (Zhou,88)) or to a subset thereof (Ni,85).

To reduce the overhead generated by the algorithm, we also introduced the concept of logical neighbourhood, i.e. each site is interconnected with a certain number of other sites, which we call its logical neighbours. Each site obtains information from its logical neighbours, and can also request to transfer execution of a process to them. Site subsets thus defined may be disjoint, and define a logical topology that exists on top of the physical network topology. Varying this logical topology varies the information on which the algorithm bases its decisions (full information if all sites are interconnected, partial information otherwise). Defining sets of interconnected sites (via 'gateway' sites) also enables processes to be propagated through neighbouring sites along the network.

2.2 Transfer policy specification

The transfer policy starts by examining how processes are scheduled at various instants according to the current load distribution. Two situations may arise:

- Load balancing: work load is distributed equally among all processors. Processes may therefore be rescheduled every time the load on a site varies.
- Load sharing: work load is smoothed out progressively on each individual processor. This avoids any site becoming temporarily overloaded, and a process need only be rescheduled if the load on a site becomes too high.

Although the second technique involves less overhead, we decided to balance the load on each system processor. Logically, if the scheduling algorithm is efficient, load sharing should not have an adverse effect on system performance.

Generally speaking, transfer policies are based on thresholds (Stankovic,84; Shivaratri,92). In other words, a processor transfers a process if its workload exceeds a predetermined threshold, otherwise it can receive processes itself. A relative transfer policy may also be employed (Douglis,91), whereby the level at which a processor is considered overloaded is defined with respect to the load on other processors.

The algorithm proposed employs a combination of the above techniques. A site S interconnected with a neighbouring site S_N with a lower load becomes a candidate to transfer its processes. S_N will receive if its load is, at least, a certain percentage lower than that of S.

Choosing which process to move means deciding which process is the best candidate for remote execution. Some research has been done on manual filters using a special command to indicate which process should be transferred (Folliot,92). Other research has looked at the use of transparent filtering (Svensson,90; Ju,95) based on estimated CPU time used up by a process, or the resources it requires to execute.

Our study places no restriction on remote process execution. We therefore assume that any process can be transferred to and executed at another site. The process that is normally sent to execute at a remote site is the one which causes processor load to increase in the first place, thus switching it to a 'transfer' state. However, to ensure that the algorithm remains stable (i.e. to avoid a process migrating through the network in vain without ever terminating (Stankovic,85)) a process returned to a site where it already been executed must finish executing on that site.

2.3 Location policy specification

The final step in a dynamic scheduling algorithm involves selecting the processor to be allocated to the process identified by the transfer policy in the preceding step.

Again, a centralized policy is considered unsuitable for the reasons already discussed (§2.1). By employing a decentralized policy, a receiving processor can be chosen randomly (Random in (Zhou,88)) or cyclically (Wang,85), i.e. without knowledge of processor status. Where processor selection is predicated on a certain degree of prior information (full or partial), the entities applying the location policy must co-operate. As we shall see in section 3.3.3, we plan to use a scheme whereby sending and receiving processors negotiate before a process is effectively scheduled. This method is also used in (Bryant,81; Stankovic,84).

With the broad outline of the dynamic scheduling algorithm proposed in this paper now established, we shall look at the model adopted to implement it.

3 A TWO-LAYER MODEL

We have seen that the algorithm must be applied in a decentralized manner. This is achieved through a set of entities distributed throughout the system. We shall call one of these entities the 'agent', in the sense given to this term in distributed artificial intelligence (Ferber,88).

Deciding to execute two communicating processes at two remote sites involves a trade-off between the performance gains achieved by executing them in parallel and the cost generated by their inter-communication. While many static scheduling algorithms attempt to reduce inter-process communication costs (Billionnet,89), very little consideration has been given to this problem when designing dynamic algorithms (Stankovic,84; Folliot,92). This is one of the main aims of this study.

Rather than use an objective function based on run times and communication costs, we chose to express these two parameters separately using a two-layer model:

- a 'system agent' (SA) is associated with each site and manages that site;
- an 'application agent' (AA) is associated with each application on each site and handles communications within that application.

3.1 System agents

The purpose of a SA is to reduce the workload on its site by transferring execution of certain processes to its logical neighbours. It does this by evaluating its own site load and the load of its logical neighbours. Co-operation between SAs in the network balances workload over the various network sites.

3.2 Application agents

The purpose of an AA is to reduce IPC costs within the application, by co-operating with other AAs associated with the application. It does this by tracing calls to primitives for inter-process communication and compiling process communication statistics.

3.3 Interaction between agents

SAs exchange information with each other on the status of their associated site. In attempting to reduce workload on their site, SAs tend to disperse processes belonging to the application over the network.

An application's AAs also exchange information on inter-process communication. In attempting to reduce IPC costs, they tend to keep processes as close to one another as possible.

As system and application agents often reach conflicting scheduling decisions, they have to co-operate, sometimes negotiate, to resolve contention. This requires a third level of communication between site SAs and local AAs.

We shall now look at how such co-operation is achieved within each of the three algorithm policies.

Co-operation in information policy

The SA alone decides where processes at a site S shall execute. To ensure this decision takes account of inter-process communication constraints, the local AAs must have a bearing on the SA's choice. But a process P waiting for a CPU time slice to be allocated to it is unable to send or receive messages. Further, execution of other processes waiting for messages from P is also slowed down. To speed up execution of P it must therefore be transferred to a site with a small workload. An SA must therefore eliminate any processes executing locally or prevent remote processes from being executed locally.

The solution adopted therefore consists in increasing site load virtually by including inter-process communication in load computations. An AA does this by giving a 'weight' to each of the processes it controls, according to how communication-intensive it is. This weighting is then taken into account by the local SA when computing its own site load.

Co-operation in transfer policy

An SA detects when its site is overloaded. It can then decide alone which process to move; but it may first consult AAs at its site if required. An AA responds by indicating which process would most reduce communication overhead within the application if it were transferred. The SA then makes its choice on the basis of all responses received.

Co-operation in location policy

To improve the stability of the algorithm, a negotiation phase is introduced between the SAs of the sending and receiving sites. The receiving site commits itself to execute a transferred process on arrival. This means that:

- the process will not be rejected on arrival at the receiving site;
- the receiving site will not become overloaded if several transferring sites select it simultaneously.

A receiving site thus accepts a process before it is transferred and is able to make room for it before it arrives.

Where co-operation between SAs and AAs is necessary, we need to make sure that process scheduling is efficient enough to compensate for the higher cost of the algorithm, and that the latter still reduces application response times.

However, if a certain criterion proves too costly to handle, the corresponding component can be taken out to reduce the overhead generated by the algorithm. Breaking down the algorithm in this way improves its flexibility and reduces its cost. The scheduling policy can thus be based on:

- load sharing alone if SAs alone make scheduling decisions;
- a reduction in communication overhead if AAs alone make these decisions.

4 EXPERIMENTAL PLATFORM IMPLEMENTATION

In this section we shall briefly describe how the test platform is implemented. To evaluate the scheduling strategy described below, we developed a distributed experimentation prototype written in C++ to run in a Sun/Solaris 2.4 environment. This prototype is able to simulate a network of N sites and its architecture is shown in figure 1.

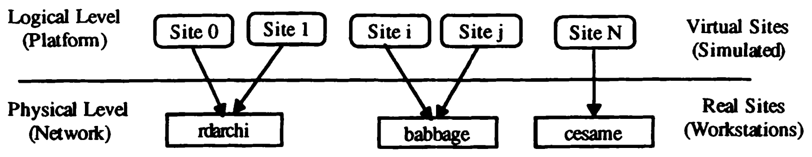


Figure 1 Prototype architecture.

A virtual site, represented by a UNIX process, is activated by the Arcadia command. The user is able to configure simulations by setting this command's parameters (number of virtual sites, scheduling strategy, etc.) and via additional configuration files. The components of a virtual site are represented by lightweight processes:

- the CPU handling processes at the site;
- the site SA;
- the AA associated with each application likely to send a process to execute at the site;
- applications initially started at the virtual site.

These various components are arranged according to the logical scheme of a virtual site shown in figure 2. We shall now look briefly at the overall behaviour of each of these components.

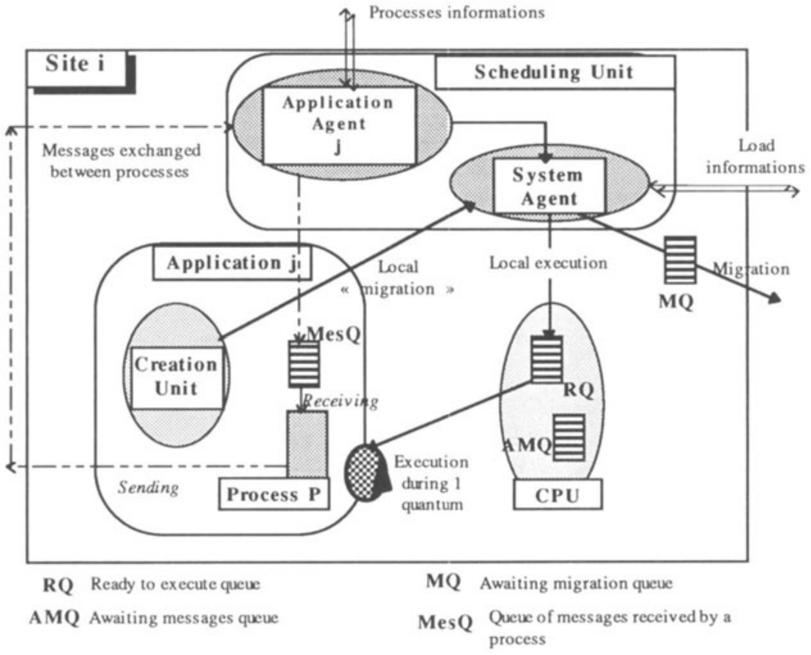


Figure 2 Logical scheme of a virtual site.

4.1 Behaviour of the components of a virtual site

Applications
An application is 'monitored' and managed by a certain number of AAs. It first has to activate its agents, and as an application cannot migrate (only processes can be moved) it has to go through an intermediate SA at each remote site to activate them. The application then creates a 'parent' process from which all application processes will be created, which waits for its 'child' processes to terminate before terminating itself. We thus have a hierarchy in which each child process can also create its own child processes, with the application at the top of the pyramid. The application therefore waits for all its processes to terminate before terminating itself.

Processes
Unlike all the other entities in the prototype, a (virtual) process is not represented by a thread. Indeed, processes must be able to migrate and it would be inexpedient to build a tool for real process migration, even for lightweight processes. Process behaviour is therefore described via a subroutine that simulates process execution. The code run by a process is replicated at each network site. Thus, process migration between two sites amounts to transferring the data needed for a process to resume execution at the remote site and the cost of this migration is the cost of the transformation of data into messages and conversely.

CPUs
Local processes are executed on each CPU in a time-sharing environment. The CPU does this by placing processes in three queues: ready to execute, awaiting message, and finished. It then selects the process at the head of the ready queue at the start of each CPU cycle and allocates it a time quantum.

Agents
We can think of an agent as an entity that reacts to events (the load on a neighbouring entity falls) and takes scheduling decisions (transferring a process to that entity). It detects local or remote events directly by analyzing its own data, or indirectly on the basis of messages it receives and processes according to their priority (urgent, rapid, or normal) and then on a first-come first-served basis.

SAs and AAs do not detect the same events and react differently. For this reason, we shall distinguish between them in our analysis of how the algorithm is implemented in our prototype.

4.2 Implementation of Arcadia scheduling strategy

Information policy
Statistics on a process P indicate the number of messages it has passed or received during an interval of S seconds, the amount of data it has passed or received in that interval, and the identity of its companion processes.

Site workload is represented by the sum of the weight values attached to the processes at a site at a given instant. The weight of a process P depends on the number of receiving processes under its control over the last S seconds. Priority is thus given to processes that are passing messages. Currently, the weight of P is equal to the number of receiving processes P possesses but its calculation should evolve to take into account other criteria.

An SA manages a vector storing information on the load of neighbouring sites. This vector is updated each time a message is received conveying the new load of one of these sites. An SA communicates with its logical neighbour sites whenever its site load increases (when a process is created, execution of a remote process is accepted or the weight of a process increases) or decreases (when a process finishes executing locally, it can execute at another site or its weight decrease).

Transfer policy
An SA considers that its site is overloaded when its load exceeds that of its logical neighbours, and decides that a process needs to be moved. Generally speaking, it is the process at the root of the overload that is moved to execute remotely.

The SA and AA negotiate if the number of processes ready to execute at a site S exceeds a certain threshold, and if the load of S is above the average load of its logical neighbour sites.

An AA calculates the reduction in communication costs achieved for each process P it controls that is sent to execute at the site selected by the SA. This calculation is based on the amount of data exchanged by P during the most recent time interval. Performance gains could also be estimated by attempting to forecast subsequent process communications. An AA then signals to the SA which process should be moved to obtain the highest gain.

Location policy
The location policy, working in tandem with the transfer policy, locates the neighbouring site with the lowest load below a threshold proportional to the load of the transferring site. Our policies are thus relative.

The SA at the receiving site accepts the process to be sent providing its execution will not push its own load above that of the sending site, and above the predetermined threshold.

We chose to apply a threshold to control site selection in order to allow for a certain degree of uncertainty with regard to network status, and thus to render the algorithm more stable.

5 ANALYSIS OF SOME EXPERIMENTAL RESULTS

We shall now evaluate the results obtained with our prototype to ascertain the efficiency of our scheduling strategy. This will lead us in turn to consider to what type of applications the algorithm is best suited.

The results discussed here were obtained by running simulations of a network of 8 virtual sites on a four-processor Sun Sparc-20 workstation. An application comprising a certain number of interacting processes (300, 2, 30, 8, 350, 10, 30 and 260, respectively) executed at each site.

5.1 Performance measurements

Each site simulated by the prototype returns results to an associated file. These results are used to obtain performance measurements enabling the impact of the scheduling algorithm on the applications tested to be assessed. Performance criteria measured were:

- overall performance gains provided by the algorithm, i.e., the difference between the response time obtained, for all the applications, with and without the scheduling strategy;
- distribution of workload over the sites during the simulation. This may be measured in terms of the number of processes ready to execute at a site over time, or the time spent by each site executing processes;
- the amount of negotiations between sending and receiving SAs that end with a process being effectively transferred, in order to evaluate the location policy.

5.2 Algorithm performance

To measure the influence of some parameters of the algorithm, we have set the value of the others and we varied the value of the considered parameters. This method has been applied for all the parameters of the algorithm. The results reported below are averages obtained from several simulations using the same parameters.

Varying the logical topology

In the first applications we tested, processes communicate in pairs, therefore, the weight has no influence on the performance (all processes have the same weight) and we varied the logical topology of the 8-site network. The threshold applied by the location policy was 30% of the load of the sending site. The different logical topologies used are shown in figure 3.

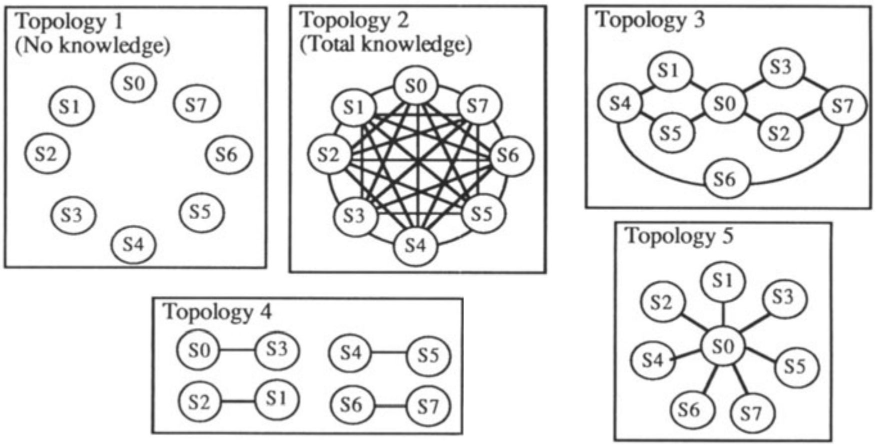
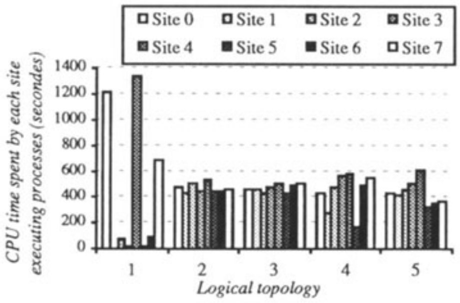


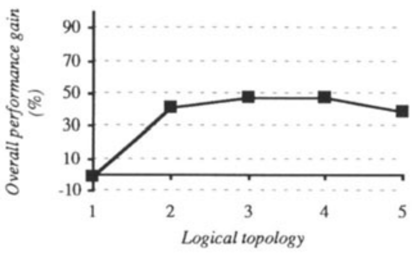
Figure 3 Logical topologies used.

Two of these topologies were diametrically opposite:

- Topology 1: each site had no logical neighbour sites. No remote scheduling was therefore possible, but the fact that sites attempted nevertheless to move processes resulted in an overall performance drop of roughly 1% with respect to when the algorithm is not applied (figure 4a).
- Topology 2: all sites were able to interact. Although this topology enabled load balancing between sites (figure 4b), it did not achieve the highest overall performance gain. This is because the information policy created a lot of overhead, since the algorithm was slowed down by the large number of information or control flow messages.



(a) Load distribution over sites.



(b) Overall performance gain provided.

Figure 4 Varying the logical topology.

To reduce this overhead, we reduced the number of logical neighbours connected to each site and adopted topology 3. Load balancing was still efficient, but this time the overall gains achieved with the algorithm increased by roughly 6%. Practically the same result was obtained with topology 4, in which pairs of sites were able to interact, but load balancing was clearly not as good. This was due to the deliberate link we introduced between a site running a 'small' application and a site supporting a 'large' application. This allows the bigger applications to load off onto the smaller applications in expedient fashion. They do not transfer sufficient load to balance out overall load completely, but enough to achieve a good overall level of performance made easier by using a less costly information policy. It is likely that the impact of these last two topologies would be greater on a larger network, as total knowledge would generate a prohibitive cost for the algorithm.

By adopting topology 5 and allowing a single site to interact with the entire network, we found that this 'gateway' site becomes overloaded with information messages and, although it can distribute load over the network, is therefore less efficient than the other topologies already described.

Varying the logical network topology and, therefore, the knowledge on which our agents rely, shows that SAs and AAs can achieve the overall goal of improved response time with only a partial view of their environment, i.e. a restricted number of logical neighbours.

Linking underloaded and overloaded sites would not appear feasible, however, as their status cannot be predetermined. In choosing a logical topology, we are therefore forced to make a trade-off between the following requirements:

- sites must have a limited number of logical neighbours to limit overhead due to the information policy;
- sites must have a large enough number of logical neighbours to be able to load off excess processes;
- certain sites must serve as gateways between subsets of sites, so that processes can be disseminated through the network.

Our research suggests that a good solution would be to adopt an initial logical topology capable of evolving dynamically in response to site loads and application behaviour. This problem has also been discussed in (Kremien,93).

Bearing of process weight and process migration

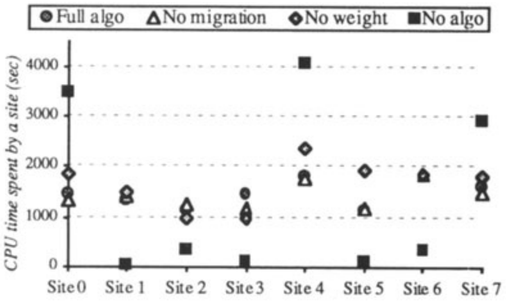
One seeks now to measure the influence of the weight. To do this, some processes of an application passe messages to several other processes thaht only receive. Working on the basis of the same load imbalance, we shall now compare cases where:

- the full algorithm is applied;
- process migration is forbidden, meaning that a process can only be moved when it is created;
- process weight is not taken into account in site load calculations.

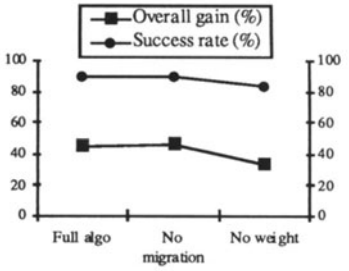
These cases are indicated by the legends 'full algo', 'no migration' and 'no weight' in figure 5.

Process weighting has an affect on scheduling efficiency, and no weighting at all reduces the performance gains achieved on all applications and the success rate of inter-site negotiations by around 10% (figure 5b). A load imbalance tends to persist when weighting is not used (figure 5a). It would therefore seem that weighting a process according to the communication overhead it generates is a reasonable assumption to work on.

We can also see from figure 5 that, in our simulation at least, no benefit is gained from moving a process during its execution. Processes transferred during execution are those whose weight has just changed, meaning they are processes that have already used up a certain amount of CPU time. The execution time remaining is probably too short for remote execution to be of any benefit to them, as the overhead generated by transferring them outweighs any performance gains obtained. Failing to put a limit on remote process execution can therefore lead to perfectly needless process migration.



(a) Load distribution over sites.



(b) Overall performance gain provided and success rate of negociations.

Figure 5 Effects of the variation of the algorithm.

6 CONCLUSION

The algorithm described here is a completely decentralized, dynamic scheduling algorithm whose unique feature is its ability to take into account inter-process communication. The algorithm is split into two parts, thus generating two types of entities to apply its policies: 'system agents' and "application agents'. These agents co-operate and negotiate with each other, i.e.:

- co-operation between SAs allows load sharing between sites;
- co-operation between AAs reduces process communication costs within an application;
- co-operation between SAs and AAs allows load sharing and reduces IPC costs in all applications.

Process execution time and communication overhead are expressed by 'weighting' each process. The weight of a process depends on the number of potential receivers of its messages, and can only be calculated by an AA. SAs use these weight values to calculate their site load.

We developed a prototype capable of running applications over a network of sites to test our scheduling strategy. A lot of improvements still need to be made before we can consider that the scheduling strategy we have adopted is really effective, and before identifying the types of application to which it would be best suited. However, results obtained have justified certain conceptual hypotheses and opened up numerous research prospects:

- Allocating a weight to each process to factor in communication overhead is a viable working hypothesis. However, weight computation needs to be improved and further tests will be required.
- It ought to be possible to make the scheduling algorithm more adaptable by distributing it over the network through the use of agents. An agent could be allowed to request or load off work depending on the perceived status of its environment.
- Process migration does not appear essential for the applications we tested, unless processes executing remotely are filtered. However, migration would certainly prove useful where the risk of site overloading is high, or to provide fault tolerance.
- Overall performance gains vary from application to application, but the 45% to 60% improvement obtained is satisfactory. However, our results were enhanced by deliberately creating large load imbalances between sites.

7 REFERENCES

Bernard, G. Steve, D. and Simatic, M. (1993) A survey of load sharing in networks of workstations. *Distributed Systems Engineering,* **1-2**, 75-86.

Billionnet, A. Costa, M.-C. and Sutter, A. (1989) Les problèmes de placement dans les systèmes distribués. *T.S.I.*, **8-4**, 307-337.

Bryant, R. and Finkel, R. (1981) A stable distributed scheduling algorithm. *Proc. of the 2nd ICDCS* 314-323.

Douglis, F. and Ousterhout, J. (1991) Transparent process migration: design alternatives and the Sprite implementation. *Software-Practice and Experience*, **21-8**, 757-785.

Ferber, J. and Ghallab, G. (1988) Problématique des univers multi-agents intelligents. *Journées Nationales du PRC I.A. Toulouse* , 295-320.

Ferrari, D. and Zhou, S. An empirical investigation of load indices for load balancing applications. *Proc. of PERFORMANCE'87* , 515-528.

Folliot, B. (1992) *Méthodes et outils de partage de charge pour la conception et la mise en œuvre d'applications dans les systèmes répartis hétérogènes.* PhD Thesis - Université Pierre et Marie Curie - Paris VI.

Ju J. Xu, G. and Yang, K. (1995) An intelligent load balancer for workstation clusters. *Operating Systems Review* , **29-1**, 7-16.

Kremien, O. Kramer, J. and Magee, J. (1993) Scalable load-sharing for distributed systems. HICSS-26.

Krueger, P. and Chawla, R. (1991) The Stealth distributed scheduler. *Proc. of the 11th ICDCS*, 336-343.

Kunz, T. (1991) The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, **17-7**, 725-730.

Litzkow M.J. Livny, M. and Mutka, M.W. (1988) « Condor: A hunter of idle workstations. *Proc. of the 8th ICDCS*, 104-111.

Ni, L. M. Xu, C-W. and Gendreau T. B. (1985) A distributed drafting algorithm for load balancing. IEEE Transactions on Software Engineering, **11-10**, 1153-1161.

Nutall, M. (1994) A brief survey of systems providing process or object migration facilities. *Operating Systems Review*, **28-4**, 64-80.

Shivaratri, N. Krueger, P. and Singhal, M. (1992) Load distributing for locally distributed systems. *IEEE Computer*, **25-12**, 33-44.

Stankovic, J. and Sidhu, I. S. (1984) An adaptative bidding algorithm for processes, clusters and distributed groups. *Proc of the 4th ICDCS*, 49-59.

Stankovic, J. (1985) Stability and distributed scheduling algorithms. IEEE Transactions on Software Engineering, **11-10**, 1141-52.

Svensson, A. (1990) History, an intelligent load sharing filter. *Proc. of the 10th ICDCS*, 546-553.

Wang, Y-T. and Morris, R.J.T. (1985) Load sharing in distributed systems. *IEEE Transactions on Computer*, **34-3**, 204-217.

Zhou, S. (1988) A trace-driven simulation study of dynamic load balancing. IEEE Transactions on Software Engineering, **14-9**, 1327-41.

Interactive testing tool for parallel programs

H. Krawczyk and B. Wiszniewski
Technical University of Gdańsk, 80-952 Gdańsk, Poland
Fax: (48)(58)416132, Email: {hkrawk, bowisz}@pg.gda.pl

Abstract
The paper presents a model for structural testing of parallel software defined as a graph consisting of interconnected node objects and a tool STEPS implementing it. The number of test cases required by STEPS is much lower than with the use of traditional flow-graph representations. The tool facilitates systematic testing procedures based on the concept of communication events and test windows. Various testing scenarios may be defined for windows at several levels of abstraction, with regard to specific classes of errors, structural testing strategies, internal system states and external environment settings.

Keywords
Structural testing, communication events, parallel control flow

1 INTRODUCTION

Developments in parallel programming languages and implementation platforms, along with raising expectations and needs of software users, have created new challenges for software testing. Various testing strategies and tools have been invented to assist users in software testing (Lutz, 1990), but they still seem not to be effective enough. Only a few of them can create test data and provide analysis of expected results. We classify them into three categories: path-wise test data generators PTDG (Therenod-Fosse and Waeselynck, 1993), data specification systems DSS (Korel, 1990), and random test generators STG (Maurer, 1990). Specifically, for testing parallel programs they may utilize several standard techniques including: *static analysis* (Taylor, Levine and Kelly, 1991), *deterministic execution* of parallel programs (Tai, Carver and Obaid, 1991), or *controlled execution* (Damodaran-Kamal and Francioni, 1994).

PTDGs are based on a program graph, where a path is chosen through a graph and is symbolically executed in order to determine a *path condition*. Path condition is a predicate that defines a subset of a program space called a *path domain*. An *input point* is a data from a path domain that can cause this path execution. Finding of input points for individual paths is not easy, and requires a capability of solving sets of equations.

DSSs generate test data from a language describing input data being prepared by a user. Examples include grammatical descriptions, finite state machines or special input (test script) files (Hoffman and Strooper, 1991). Unfortunately, DSS test data can exercise relatively small percentage of program code.

STGs combine information provided by a program model with a random test generation. They rely on two parameters: *input distribution*, and the number of *program executions*. Two approaches are possible: *analytical*, where input parameters of path conditions are generated at random, and *empirical*, where statistics on various activation of program run-time code elements are collected to create a suitable input profile. One important technique is *mutation analysis* (DeMillo and Offutt, 1991), where errors are modeled as random and small syntactical deviations of a program text from its hypothetical correct version. Test data are intended to detect incorrect (mutant) versions of a program under test.

In this paper we introduce a tool for Structural TEsting of Parallel Software (STEPS). It considers processing of data by interacting parallel processes (tasks) with regard to their relative timing. In practice a number of required paths and input points involving parallel execution can grow rapidly. We propose a *test window* approach, which concentrates on testing sequential segments of a parallel program as well as specific sets of communication actions forming *communication events* (Szczerba and Wiszniewski, 1995; Krawczyk and Wiszniewski, 1996). Such a window allows users to create different *testing scenarios*, and to analyze programs on different levels of abstraction.

The STEPS tool combines several ideas from the tools mentioned above. It utilizes symbolic interpretation to determine path conditions. Input points for individual segments are selected by users in an interactive way supported by Prolog interpreter; upon returning a condition the user may repeatedly choose input points that the interpreter uses next in an attempt to resolve or simplify a related condition. The tool uses finite state machines to describe and analyze behavior of processes during communication events (Krawczyk and Wiszniewski, 1995). This analysis is also based on Prolog interpretation. Finally the tool is able to collect data on program execution that are relevant for designing testing scenarios. It can also override dynamically (during program execution) values of variables being returned by function calls and I/O statements, what provides a limited 'dynamic' mutation capability of a program under test.

The most important feature of STEPS is that, unlike most of the tools mentioned above, it uses static and dynamic analysis jointly and enables deterministic as well as controlled execution.

The tool has been implemented for the Copernicus-SEPP project (Winter and Kacsuk, 1994) using PVM programs in C. Initial results obtained when applying STEPS to a set of realistic application programs are promising and indicate that structural testing of parallel programs can reasonably involve interactive path analysis.

2 PARALLEL SOFTWARE MODEL

STEPS views parallel programs as sets of event-driven activities that are independent units communicating through a set of well defined *interface access points*. Interface between parallel system components involves synchronization and exchange of data. Rules for communicating processes are established by *protocols*, which control ordering of selected events. STEPS distinguishes nodes in a system control flow graph associated with *internal actions* of individual processes from nodes associated with *external actions* that deal with interprocess communication (Szczerba and Wiszniewski, 1995). Relative ordering of nodes associated with internal actions belonging to different threads of control is irrelevant for STEPS, as these objects are truly independent. On the other hand, external actions dealing with interprocess communication involve a relatively small subset of events. Moreover, for any single external action (access point) in any process there is one related protocol class; any such action can always be associated through its protocol class with a specific set of external actions (access points) of other system processes.

The main feature of our model and the tool is the independence of communication events. It has been established and maintained by STEPS using *object-oriented* technology. It can involve either *top-down* design of a parallel program control flow structures using objects provided by the model before generating any actual code (Krawczyk and Wiszniewski, 1995), or *reverse engineering* of the existing code using static concurrency analysis to get the model objects, what in the case of this paper.

Actions associated with the same communication event form a group ordered by the corresponding protocol and have to ensure that the transfer of data objects between processes follows semantic rules of the relevant communication actions. Actions from different events are independent, since they manipulate different data objects associated with separate access points in component processes. Therefore, communication events cannot influence each other's internal ordering of actions. The same applies when internal actions are interleaving with external actions; internal actions are able to 'engage' some control flows in particular communication events, but cannot affect ordering of actions 'within' the individual event protocol. Owing to this, numerous combinations of interleaving external actions that belong to different events, as well as combinations of interleaving external and internal actions are in most cases irrelevant for analysis supported by STEPS. This has been shown formally using Kleene's technique of λ-automata (Szczerba and Wiszniewski, 1995).

Encapsulation of protocols eliminates the problem of combinatorial explosion of a parallel system viewed as a collection cooperating FSM's, since it constitutes for STEPS a sum of FSM's representing component processes, rather then their product; a sum machine is much simpler to analyze and understand than a product machine since the former has the set of states of smaller cardinality, i.e., as of the union of component processes' states rather then the Cartesian product of such sets (Szczerba and Wiszniewski, 1995) .

The model adopted by STEPS aims to retain the intuitive simplicity of flow-graphs, to capture the ability of Petri nets to represent multiple flows of control, and to provide an FSM-like mechanism for ordering program statements related to state transitions (Krawczyk and Wiszniewski, 1995).

Control flow of active processes can be represented in C++ by specially defined objects forming the `Token` class:

```
class Statement;
class Token {
   Statement condition; // up-to-date path condition
   Statement *memory; // up-to-date path memory
};
```

Individual `Token` objects advance through a parallel program and collect information that is specific to the path being followed. A vector (array) of all `Token` objects represents a *global state* of a parallel program.

A parallel program is modeled by multi-flowgraph *G(N,A)* using graph nodes that form a specific hierarchy of objects *N* allowing parallel control flow tokens. Any object from *N* has in general multiple entries and multiple exits connected by arcs from *A*, as specified in Figure 1. We schematically represent there each `Node` object by an icon, as well as specify the relevant class hierarchy using the C++ notation.

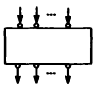
Nodes may be connected to one another. i.e., any exit of one `Node` object may be connected to any entry of another or the same `Node` object; we mark this in our model by labeling each `Node` exit with the `Action` labeling the respective entry to which this exit is connected. Labels of entries are unique, i.e., there are no two entries with the same label at any `Node` object. In general there is no precedence of connected nodes, as one node may be connected to many nodes by the respective entries (exits). Entries of a single `Node` object may be associated with different control flow tokens, therefore each such object may be entered in parallel by more than one token. Token objects, after leaving one `Node` object may spread over the system and visit many `Node` objects in parallel. Note the straightforward relationship of our model to the Petri net model: independent *tokens* represent multiple control flows, nodes are *transitions* while node entries and exits are *places*.

Visiting of `Node` objects by control flow `Tokens` in a sequential program graph normally represents execution of program statements (processing nodes) or evaluating predicates (decision nodes). Figure 1 extends this concept by specifying the respective programming constructs as either `Assignment` or `Conditional` objects, which encapsulate two specific types of `Statements`. One is an 'expression-statement', and the other is a 'predicate-statement'; they are any correct (language syntax specific) string of characters, and they may involve identifiers representing program variables as well as function calls.

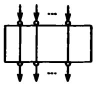
`Assignment` and `Conditional` objects allow for multiple control flow tokens to enter and/or exit their instances in parallel. At each entry of such a node incoming token causes the evaluation of the respective expressions (predicates). Note that a node entry that is not connected to any node exit represents a *creation* (instantiation) point of some process. Similarly, a node exit that is not connected to any node entry represents a termination (destruction) point of some process. Instantiation and destruction points may also involve evaluation of relevant expression or predicate `Statements`.

If `Assignment` or `Conditional` objects do not involve multiple control flow tokens, then they correspond to ordinary nodes of sequential control flow graphs, i.e., they belong to `Expression` or `Decision` objects.

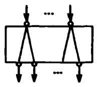
Multiple control flow tokens constitute just one feature of *G(N,A)*. Another feature is the interaction of tokens at selected points of component programs, what is due to communication statements. Owing to this feature, any `Node` object that involves parallel tokens can provide internal 'processing' of incoming tokens in order to determine what particular tokens can exit the node. Therefore such a class of nodes has properties of the `Conditional` object. It also has properties of the `Assignment` object, since interaction of tokens may result in some data objects (values) being transferred between processes. This is handled by the `CommunicationEvent` class of node objects. Internal processing of control flows is performed by finite state protocol machines, or `FSM` objects, which determine a specific order of executing communication actions involved in the corresponding event.



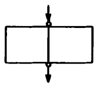
```
class Vector; class Action;
class Node {
  public:
    int N; // no of entries
    int K; // no of exits
    Action *entries; // entries labeled by current actions
    Action *exits;   // exits labeled by successor actions
    Vector initial;  // entries holding incoming tokens
    Vector final;    // exits holding outgoing tokens
};
```



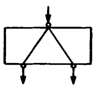
```
class Statement;
class Assignment: public Node {
  public:
    Statement *expression; // assignments at exits
};
```



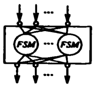
```
class Statement;
class Conditional: public Node {
  public:
    Statement *predicate; // predicates at exits
};
```



```
class Expression: public Assignment {
  public:
    const static int N=K=1;
};
```



```
class Decision: public Conditional{
  public:
    const static int N=1;
};
```



```
class FSM;
class CommunicationEvent: public Conditional, public Assignment {
  private:
    FSM **machines // array of lists of machines at each entry
};
```

Figure 1 Node objects used by STEPS.

Based on the classification of nodes developed so far one can view a set of sequential programs running in parallel as a set of interconnected `Node` objects. Paths through *G(N,A)* form `Thread` objects representing individual processes:

```
class Thread {
  private: Node *nodes; // array of Node pointers
};
```

A parallel system can be finally constructed from a set of `Thread` objects as:

```
class MultiThread {
  private: Thread *system; // array of Thread pointers
};
```

2.1 Levels of representation

The model in Figure 1 enables analyzing parallel programs at three different levels. At its highest level *G(N,A)* is similar to Lamport's *space-time diagrams* (Krawczyk and Wiszniewski, 1995; 1996), with vertical lines representing processes and horizontal arrows representing communication events. At its lowest level *G(N,A)* groups sequential program statements into graphical segments. The middle level is similar to the Petri net model, with a hierarchy of transitions (`Node` objects) shown in Figure 1.

STEPS views parallel program execution as progressing of independent `Token` objects along paths containing `Node` objects. `Token` objects visit `Communica-tionEvent` objects to be 'processed' by protocol machines encapsulated by individual instances of the latter. Encapsulated machines determine for all `Token` objects 'arriving' at entries of the related `CommunicationEvent` object what `Token` objects are allowed to leave the latter and at which of its exits.

2.2 Analysis of events

Representation levels in STEPS enable event-driven testing at the top-most level, as well as testing and symbolic debugging at the program statement level. Careful analysis of application protocols in various parallel programming environments indicates that the number of different protocol machines is reasonably low. For example, typical network oriented application platforms based on message passing, like TCP/IP (Comer, 1991) or PVM (Geist, *et al.*, 1994) distinguish just two types of communication statements for sending: one is *point-to-point*, when a single process sends a single message to another single process, and another is *broadcast*, when a single process sends a single message to a group of processes. Communication statements for sending may be either *blocking* or *nonblocking*. The blocking send forces sender to wait until receiver is ready to read a message being sent, while the nonblocking send puts a message to some buffer regardless of the state of the receiver. TCP/IP provides both types of the send communication statement, while PVM just the nonblocking one.

There are also two types of communication statements in TCP/IP and PVM for receiving, either *blocking* or *nonblocking* mode. In any mode, a receiver may want to receive a message from a specific sender or just any sender.

This defines three basic classes of application protocols: *one-to-one*, *many-to-one*, and *one-to-many*. If we distinguish blocking and nonblocking send statements, as well as blocking and nonblocking receive statements there are *twelve* kinds of protocol machines in total.

A protocol machine has one initial state, one final state and one internal state per each relevant pair of 'matching' send-receive communication statements. Such an internal state is simply labeled as 'communication engaged' and used by STEPS for reporting on the internal status of the related interprocess communication for the external observer. The set of internal states of all protocol machines encapsulated by the related instance of the `CommunicationEvent` object determines all possible sequences of communication actions constituting an event under test.

The relationship between predicates and expressions of the `Communication-Event` inherited from the `Conditional` and `Assignment` objects, and states of the relevant set of protocol machines is used by STEPS to facilitate standard symbolic interpretation. A predicate `Statement` associated with each respective exit of any `CommunicationEvent` object is a Boolean function on a vector indicating an initial state of the `Node` object; it yields 'true' for its related exit if for a given vector of active entries a respective token is allowed to proceed to that exit. An expression `Statement`, also associated with selected exits of `CommunicationEvent` objects is a conditional assignment with a predicate being a Boolean function on a vector indicating a final state of the `Node` object. It simply tells what is a data value being transferred from the message buffer upon exiting a particular `Node` object in a given state. Transformations of the `Node` object's state from the initial to the final (state) vector is determined by internal connections of a particular `FSM` object.

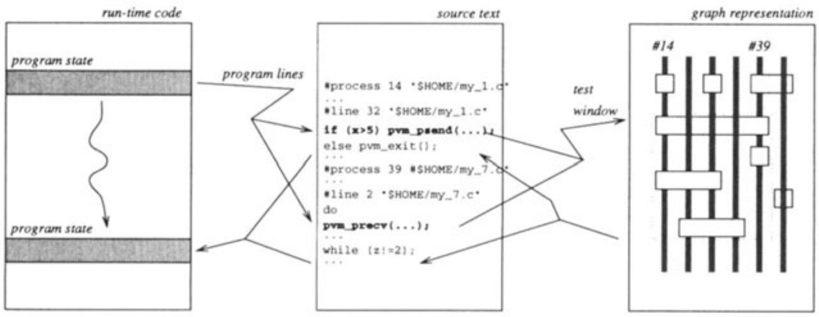
When `Token` objects visit `Node` objects they follow specific 'paths' through the related `MultiThread` object. Symbolic interpretation of a path through the latter produces an expression determining relevant input points *and* timing relations between interacting tokens at respective nodes along that path. Such a 'path expression' constitutes a TEsting Scenario Script (TESS) that provides a basis for deterministic as well as controlled run-time code execution.

3 TESTING SCENARIOS

The model of parallel programs used by STEPS allows us to determine different testing scenarios. Testing scenarios may refer to the entire parallel system under test or just its fragments forming *test windows*. A test window is defined simply as a set of `CommunicationEvent` objects between a top and bottom segments of a window frame. Top and bottom segments indicate communication events that occur upon entering and leaving a window by a specific set of parallel processes. A test window width defines processes being involved in a window.

The idea of test windows is based on the standard notion of breakpoint traps used by traditional debuggers. Processes of interest are expected to reach eventually a window top, and then proceed towards its bottom. While inside a window, processes may be stopped, their variables inspected, statements executed in a step-by-step mode, etc.

STEPS distinguishes three levels of representation of parallel programs, which are respectively the *graph representation* level, the *source text* level and the *run-time code* level. Figure 2 outlines schematically the relationship between these levels and illustrates how STEPS can merge three basic categories of testing characterized before. By introducing symbolic interpretation of a program source text at the abstract graph representation level the tool enables interactive path analysis and selection of input points, i.e., utilize PTDG. Paths selected by the user have TESS scripts generated automatically by STEPS from the underlying program source text level; TESS scripts are the input to the run-time code representation level, i.e., utilize DSS. Finally, STEPS utilizes STG by visualization of the run-time code execution at the graph representation level and on-line mechanisms for user interaction from that level back to the run-time code level.



```
*process 14 '$HOME/my_1.c'
...
*line 32 '$HOME/my_1.c'
if (x>5) pvm_psend(...);
else pvm_exit();
...
*process 39 *$HOME/my_7.c'
...
*line 2 '$HOME/my_7.c'
do
pvm_precv(...);
...
while (z!=2);
...
```

Figure 2 A relationship between abstraction levels in STEPS.

Figure 2 shows how important it is to find correspondence between levels of a parallel program representation. STEPS assumes that static and dynamic analysis may be done in different orders. We have created functions capable of distinguishing lines and their corresponding states in the runtime-code. These functions will be characterized briefly in the next Section. Below we want to emphasize the need for test windows and their representation at various levels. We can define three types of testing scenarios:

1. A set of windows in a source (run-time code) to check on a given sequence of events; sizes of windows can be arbitrary, but in many cases users may want to restrict windows to single vectors of breakpoint traps, i.e., windows of height = 0.
2. Single (and independent) windows to check on communication between selected processes; in this case users may work interactively to attempt various input (test) points for the assumed order of communication events.
3. Blocks of code (sequential nodes) to check on sequential parts of parallel programs; these operation involve symbolic debugging using standard tools, like the GNU 'gdb' debugger.

4 THE STEPS TOOL

The STEPS tool performs a number of processing activities including retrieval of information from the program source code in order to construct it's respective multi-flowgraph, visualization of a program control structure enabling users to specify test windows, symbolic interpretation of program paths assisting users in designing testing scenarios, and dynamic execution of tests enabling interactive debugging.

There are four major functional components presented schematically in Figure 3:

- A test *window user interface* (WUI), to interact with static analysis of a program text, selection of program paths and their dynamic execution.
- A *static analyzer of PVM text* (SAPTE), to identify communication events using reachability analysis and determining quality parameters.
- An *interactive data test generator* (IDATEG), to determine execution conditions of paths and to assist users in finding suitable data for program execution.
- A *testing scenario execution manager* (TESEM), to define and execute testing scenarios at different levels of representation (multi-flowgraph or sequential blocks).

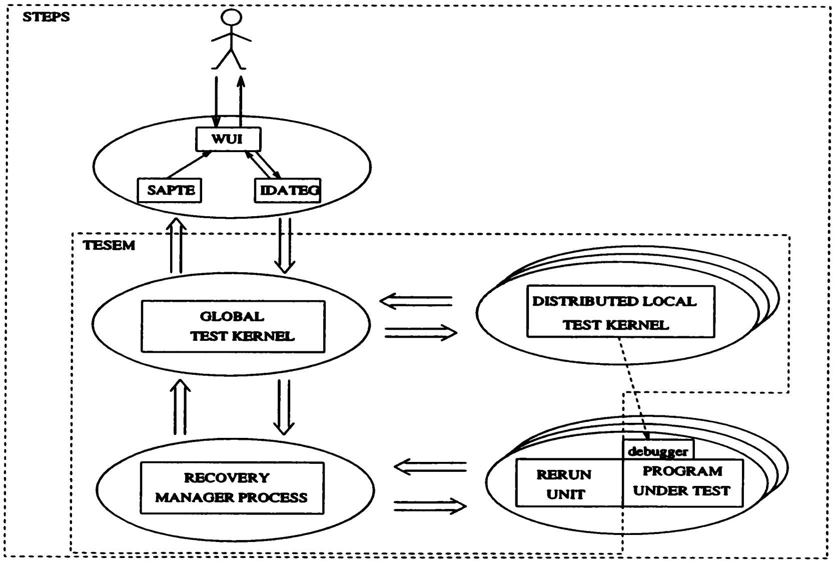


Figure 3 An overview of STEPS architecture.

Activities of these four subsystems', kinds of information being passed between them and related information repositories are shown schematically in Figure 4. It can be seen that there are two kinds of input information required by STEPS: a program input source code and user commands. An initial user command is to start *static concurrency analysis* of the program input source code. As a result of this analysis three kinds of information are retrieved from the program text: a structure of component processes' graphs, respective sets of communication actions identified as communication events, and event ordering relations. This information is stored for further processing in three respective repositories: a collection of *component graphs*, an *event table*, and a reachability tree in a form of *state transition graph*. Information retrieved by static analysis is used by interface WUI to display a multi-flowgraph structure needed by users for preparing testing scenarios. This requires information about all node connections and communication event nodes. Based on this information user can outline a desired window frame, that provides a basis for *window definition* activities.

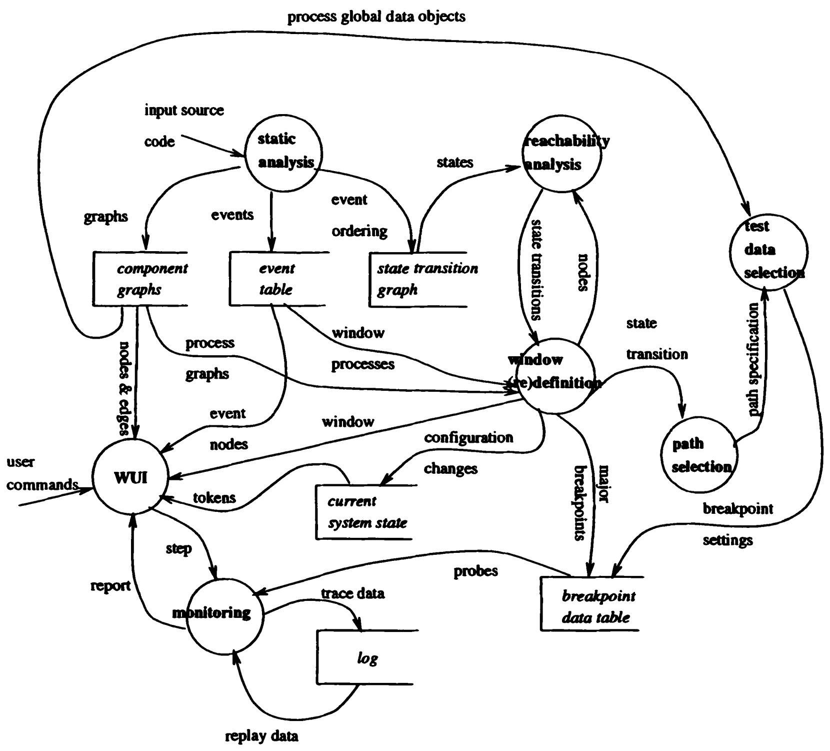


Figure 4 Data flow diagram of STEPS.

Window definition activities require finding relevant information on internal control structures of component process graphs, and determining from the structure of communication events selected for the window what are the processes going through it. Since window framing must avoid the probe effect, i.e., window processes are not allowed to communicate with other processes not going through the window, another activity, *reachability analysis*, is needed. Reachability analysis checks for selected nodes whether respective states specified by a state transition graph can be reached within a window. Corresponding state transitions are provided by reachability analysis for activities following window definition, i.e., to be utilized in *path selection*. The latter activity results in path specification to be analyzed for *test data selection*. Test data selection involves analysis of statements along specified paths and their operations on relevant process data objects; these objects are global from the viewpoint of individual paths contained in a test window. Test data are associated by TESS scripts with specific program variables in a form of breakpoint traps that specify what variables and at which points of execution have to assume specific values, and what these values should be. Breakpoint statements are considered to be *minor breakpoints*, since they mark specific points of execution at the program source text level, as opposed to *major breakpoints* that mark monitoring points at the graph representation level and are determined by window frames. Major and minor breakpoints, as well as associated test data are stored in the *breakpoint data table* being a repository used by *test manager* activities. Test manager receives step specifications from WUI and initiates execution of a program code under test using information stored in the breakpoint data table; this information specifies probes that have to be inserted into the *run-time* code. During program execution trace data are collected and stored in the *log* repository. Upon completing a step the program execution is suspended and the relevant step report is returned to WUI. Test manager may either initiate the next step or replay the previous one; in the latter case it retrieves respective replay data from the log.

5 CONCLUSIONS

STEPS implements an object oriented model that enables analysis and design of parallel programs to make them testable. This model provides a generic representation of any parallel program control flow structure based on message passing and includes multiple threads of control to capture natural parallelism of program components, external and internal actions to distinguishing relevant sequences of component processes' actions from irrelevant ones, and communication events to provide the main structuring concept for interprocess communication. The key feature of this model is its capability to encapsulate communication actions in objects constituting disjoint communication events, to provide well-defined access points to parallel processes, and to enable flexible representation of any realistic communication protocol.

A testable representation of a parallel program is provided by STEPS at several levels, including test windows, lines of source code text and execution states of a run-time code. Owing to this any structural testing strategy that uses the notion of a program control flowgraph can be utilized in developing testing scenarios; it involves designing and preparing tests, as well as evaluating test coverage.

The natural parallelism and independence of objects in *G(N,A)* is suitable for rapid prototyping of a program design: based on a source code one may attempt to build an abstract program model using static analysis and then generate a new, higher quality program. Such an extension of STEPS is currently under development using Smalltalk and C++.

6 REFERENCES

Comer, D. (1991) *Internetworking with TCP/IP Vol.1: principles, protocols, and architecture.* 2nd ed., Prentice-Hall.

Damodaran-Kamal, S. K., and Francioni, J.M. (1994) Testing races in parallel programs with an OtOt strategy, in Proc. of the 1994 International Symposium on Software Testing and Analysis, Seattle, WA, USA, 216-227.

DeMillo, R.A. and Offutt, A.J. (1991) Constraint based automatic test data generation. *IEEE Trans. on Software Eng.*, **17(9)**, 900-910.

Geist, A., et al. (1994) PVM 3 user's guide and reference manual. Oak Ridge National Lab, Oak Ridge, Tennessee, 1994.

Hoffman, D.M. and Strooper, P. (1991) Automated module testing in Prolog. *IEEE Trans. on Software Eng.*, **17(9)**, 934-943.

Korel, B. (1990) Automated software test data generation. *IEEE Trans. on Software Eng.*, **16(8)**, 870-879.

Krawczyk, H. and Wiszniewski, B. (1995) Design for testability of parallel programs, in Proc. of the 4th Software Quality Conference, Dundee, UK, 1-10.

Krawczyk, H. and Wiszniewski, B. (1996) Object oriented model of parallel programs, in Proc. of the 4th Euromicro 96 Workshop on Parallel and Distributed Processing, Braga, Portugal.

Lutz, M. (1990) Testing Tools. *IEEE Software* **7(3)**, 53-57.

Maurer, P.M. (1990) Generating test data with enhanced context-free grammars. *IEEE Software*, **7(4)**, 50-55.

Szczerba, A. and Wiszniewski B. (1995) A tool for testing communication events in TCP/IP environments, *in Parallel Programming: State of the Art Perspective ParCo'95*, (ed. E.H. D'Hollander, G.R. Joubert, F.J. Peters, D. Trynstram) Elsevier Science, 1996.

Tai, K.-C., Carver, R.H., and Obaid, E.E. (1991) Debugging concurrent Ada programs by deterministic execution. *IEEE Trans. on Software Eng.*, **16(8)**, 897-915.

Taylor, R.N., Levine, D.L. and Kelly, Ch.D. (1992) Structural testing of concurrent programs. *IEEE Trans. on Software Eng.*, **18(3)**, 206-210.

Therenod-Fosse, P. and Waeselynck, H. (1993) STATEMATE applied to statistical software testing, in Proc. of the 1st ACM Symposium on Software Testing and Analysis ISSTA'93, 99-109.

Winter, S. and Kacsuk, P. (1994) Software engineering for parallel processing, in Proc. 8th Symp. on Microcomputer and Microprocessor Applications, Budapest, Hungary, 285-293.

7 BIOGRAPHY

Henryk Krawczyk is a Professor in Computer Science at the Technical University of Gdańsk. Current research interests include dependable computer systems, software quality assurance and metrics, parallel and distributed processing. Member of IEEE.

Bogdan Wiszniewski is an Assistant Professor in Computer Science at the Technical University of Gdańsk. Current research interests include testing of sequential and parallel computer software, object-oriented programming and computer networking. Member of ACM.

10
Cerberus - a tool for debugging distributed algorithms

F. Carter and A. Fekete
Department of Computer Science, University of Sydney
Madsen Building F09, University of Sydney 2006, Australia.
email: `fekete@cs.su.oz.au`

Abstract
Distributed applications are hard to program. They are particularly prone to subtle race conditions, deadlocks, or similar errors in the underlying distributed algorithm. This paper describes a tool which can assist the designer in debugging a distributed algorithm early in the software lifecycle. The tool takes a high-level abstract description of the algorithm, and an even more abstract requirements specification; it simulates an execution until a discrepancy arises between algorithm and specification; it then assist the developer to explore backwards and forwards through the execution till the error is understood.

Keywords
Distributed programs, debugging, simulation, formal methods

1 INTRODUCTION

Programming a distributed application is even more difficult and error-prone than writing other software systems. One source of difficulty is the lack of programmer experience with the languages and tools used in coding: the domain is relatively new, and standards are still developing, so different systems are incompatible and many unnecessary obstacles are placed in the programmers' path. Whole books have been written to assist with these arcane details (Stevens, 1990). Distributed applications however differ from tasks in other domains in that even before the coding phase is reached, the design phase is often affected by fundamental algorithmic errors, which are hard to detect, and are not amenable to being fixed with small corrections. For example, in a sequential program, a common mistake in algorithm design is an "off-by-one" loop, which fails to check the last element of a sequence; this is easily detected by testing with a range of extreme inputs, and is fixed by changing the termination test in the loop. In contrast, a distributed algorthm may have a "race condition", where the error is revealed only when a particular pattern of message delays occurs, and the correction requires a completely new approach. The greater difficulty of design for distributed algorithms is clearly evident in the high rate of errors among papers written by experts and accepted in prestigous journals. As as extreme case, Knapp (1987) discusses the sad sequence of incorrect algorithms for detecting deadlock.

Because a distributed application is so vulnerable to errors in algorithm design. we believe that the development of these applications would benefit from a tool that allows the debugging of the algorithm itself, in a rather abstract early form. The designer should be able to explore the executions of the underlying algorithm intended for their system; only once a sound design is chosen would coding take place. This paper describes a tool of this sort. It is called Cerberus and a first version has been implemented.

It is important for the reader to distinguish the sort of tool we describe, which is used for debugging a distributed algorithm at the design phase, from those which can assist in debugging a deployed distributed system after the coding is completed. The latter sort of tool must itself be a distributed program, collecting information at multiple sites in a network, and attempting to determine whether or not certain global conditions are satisfied. Because remote information is always out-of-date, a debugger for distributed systems is very hard to build. Babaoglu and Marzullo summarise the theory behind these systems in chapter 4 of the book edited by Mullender (1993).

The view of software development in this paper is based on top-down refinement: the designer starts from a requirements specification of the service the system is expected to provide to its clients. As described by Fekete (1993), this service specification will generally be presented as a global, abstract, state transition system. Next the designer decides on a fundamental algorithm that will provide this service. For example, the algorithm might involve a token traversing the network, or it might be based on a replicated state machine (Schneider, 1990). This basic algorithm is described as a collection of separate abstract state transition systems, one (for each site in the network) representing the part of the system at one site, and others (such as buffers) which provide inter-site communication. In Cerberus, both requirements specification and proposed distributed algorithm are presented in a particular syntax which is based on a semantic model called Input-Output Automata (Lynch and Tuttle, 1989), which has been extensively used in research papers for describing distributed algorithms. The Cerberus tool is used to detect errors in the basic algorithm. Once no more errors are discovered, and the designer is confident in the correctness of the algorithm, the individual site components can be further refined to efficient code in a conventional programming language. This requires converting the state-transition description used by Cerberus to flow-of-control in a language like C; one must also replace abstract data structures like sets by efficient implementations.

The top-down style of development supported by Cerberus is in contrast to the more common bottom-up building of distributed applications, where the designer takes individual components that already exist, and combines them in different configurations to meet various goals (perhaps writing additional clients to make calls on the components). This bottom-up style of composition is expected by recent standards such as CORBA or Microsoft's OLE Component Object Model; it is also supported by prototype tools such as the Software Architect's Assistant (Ng et al, 1995).

Non-determinism is a key feature of distributed algorithms, and a central reason for the frequency of major errors in their design. Even though each node in the system is completely predicatble in its responses to messages, the whole system has a very large set of executions, each corresponding to a particular pattern of unpredicatable message delays. Since the system has no control over these delays, an algorithm is considered correct only if every possible execution produces the desired outcomes. The core of the Cerberus tool is to simulate one execution of the distributed algorithm (if no error is detected in this, another execution is simulated). The tool allows the designer to control the execution directly, by repeatedly selecting the next action to occur from among those enabled at the current state; alternatively the execution may develop without the designer's intervention, with appropriate randomisation controlling the pattern of message delays etc. As the algorithm's execution is simulated, Cerberus also follows the transitions in the requirements specification. An error is detected when the algorithm takes an action not allowed in the requirement (this violates a *safety condition*) or when no action is possible by the algorithm at a time when the specification can produce output (this is *deadlock*). The organisation and interface of Cerberus has been inspired by a previous simulation tool for the Input/Output Automaton formal method, called Spectrum (Goldman, 1990).

The key contribution of Cerberus, which distinguishes it from general simulation tools, lies in what happens after an error has been identified (that is, once an incorrect execution has been found). Cerberus allows the designer to explore the execution history, trying to pin down the part of the algorithm that needs fixing. Usually the algorithmic error is in a step that occurred long before the system finally took a step not allowed by the specification. For example, the detected step is often triggered by the arrival of a particular message at a node which is in a particular state; however the error might be in the decision to send that message, or in the decision to enter the particular state (or the problem might lie further back, in the sending of the message that caused the node to enter that state). Cerberus provides the ability to jump backwards and forwards through the execution that has revealed an error: for example, one can move to the most recent step of a given node.

This paper shows how Cerberus is used. In Section 2 we explain the language used to represent both the requirements specification, and the distributed algorithm. In Section 3 we explain the facilities provided by the system and how they are implemented. In Section 4 we work through a (contrived) example. In Section 5 we summarize our conclusions.

2 DESCRIBING TRANSITION SYSTEMS

The intellectual foundation for Cerberus is the Input/Output Automaton formal method (Lynch and Tuttle, 1989) invented by Lynch and her colleagues as a semantic model used in presenting and verifying distributed algorithms. The framework has been found to be widely applicable, with simple extensions to deal with time-dependent algorithms, shared memory algorithms, several different types of modularity, and even impossibility proofs. The formal method is based on representing each component as a state-transition system, with potentially infinite state space, and where transitions are named and also classified as inputs, outputs or internal steps. A collection of components can be composed, with synchronisation provided by the fact that identically-named transitions must be taken simultaneously in all components.

As described by Lynch and Tuttle (1989), the Input/Output Automaton method is semantic; the states and transitions may be described using all the techniques of mathematics. In a software tool such as Cerberus, it is essential that a fixed syntax be used to present an automaton. This section describes the language we have chosen. As an example, we give the code in Figure 1 which models a unidirectional error free communication channel. Further examples written in the language are found in Section 4.

Classes of Automata: In representing a distributed algorithm, it is usual to have many similar automata in the system; commonly, the processing at each node follows the same principles (which indeed apply no matter what the network topology). Thus in Cerberus,

```
typedef MsgType   : tup(from,val : integer;);
typedef PktType   : tup(type : integer; msg : MsgType;);
typedef PktQueue  : seq of PktType;

automata channel()
state_vars
    buffer : PktQueue;
initial  buffer := seq();
begin
    input send(channel : integer; pkt: PktType;)
    restrict
        channel == AutomataNum;
    begin
        buffer := buffer + seq(pkt);
    end

    output receive(channel : integer; pkt : PktType;)
    restrict
        channel == AutomataNum;
    pre
        if
            (#buffer != 0) -> pkt == buffer[1];
        fi;
    begin
        buffer := buffer[2..#buffer];
    end
end
```

Figure 1 An I/O Automata class to model a communication channel

we describe a generic template which we call a class of automata, and then we simulate an algorithm in which many instantiations of this class coexist, in a particular configuration.

The class name is declared in a similar method to many programming languages syntax for declaring a procedure or function, i.e an identifier with a set of typed parameters enclosed in brackets. There is also an implicit argument to the automata that does not need to be declared, this is 'AutomataNum' which provides a unique integer identifier for each automata that is included in the final simulated system. In the example above, this implicit parameter is the only parameter.

State: The Input/Output Automaton formal method allows an arbitrary, possibly infinite, state space. In Cerberus, the state space is always given as a Cartesian product, based on a collection of named, typed, state variables. Each state of the automaton is described by giving a particular value of the correct type to each variable. Because Cerberus aims to support debugging of algorithms early in the lifecycle, it is important to allow them to be described in a rather abstract style, more common in specification languages than in common programming languages. Experience in describing many algorithms shows the usefulness of abstract, complex data types, such as a set of sequences of pairs, each of which has a string and a set of integers. To cater for these needs, three type constructors were include in the language: sequences, sets, and tuples. In the example above, there is a single state variable called buffer; its value is a sequence of tuples. Each constructed type has the usual operations: for example `seq()` denotes the constant empty sequence, and `#v` denotes the number of entries in the sequence which is the value of variable `v`.

Transitions: In the formal method, an automaton can change between states in discrete steps. There is a transition relation that defines the allowable steps. There are names given to the possible steps (these names are crucial in defining the way automata interact). Each name is refered to as an action, and each action is classified either as an input (meaning that it is controlled by the environment rather than by the automaton itself), as an output (under the autonomous control of the automaton, and able to be detected by the environment), or as internal (under autonomous control, but unable to be detected by other components). In the formal model, each action may label an arbitrary set of transitions. However, it is normal for many related actions to all be used to label transitions which can occur in similar states, and for which the state after the action is determined as a function of the state before it. In Cerberus, we define parameterised action templates. The possible values of the parameters are of course limited by their types, but the algorithm designer can also provide an extra restriction. For example, the code of Figure 1 shows that each channel automaton has many input actions, one for each possible value of pkt; however, the first parameter in the action name is fixed to be identical to the AutomataNum of this component. The restriction clause is not arbitrary: an action class parameter may only be used on the left hand side of an equality operator or the left side of the "element of" operator (written with the `[=` symbol). This allows the simulator to efficiently decide which parameter values should be considered.

It is fundamental in the formal method that input actions can occur at any time as they are controlled by the environment rather than by the automaton. However output and internal actions are under local control, so these actions have a precondition, introduced by the `pre` keyword, which is a series of boolean expressions[*]. We say an action within a class is enabled when the conjunction of expressions in this section are true. Action parameters can be mentioned in this section, under the same limitations as in the `Restrict` section; these two sections jointly decide which actions within an action class will be allowed to occur in a given state.

When an action occurs, the component must change state. In Cerberus, the new state is determined by executing a body of code enclosed `begin` and `end` keywords. This code, through assignments, defines the way in which the state variables are to be updated to move from the current state to the next state when an action from this class is executed. Within the code, one can refer to any state variables, to the parameters of the action template, and also to local temporary variables (these are meaningful only within the code, and they do not keep their value in the automaton state for use in later transitions). The code should alter values of local or state variables, but not of parameters of the action and automaton. The code may also contain assignments which are executed conditionally; we use the syntax `if` *boolexp* `->` *code boolexp* `->` *code* ... `fi`. The meaning is that each *boolexp* in turn is evaluated; whichever is the first that evaluates to true; the corresponding *code* is executed[†]

[*]For convenience in translating existing algorithm models from the research literature, which are written without specific syntax, we provide syntactic sugar for implication written as `if` *boolexp* `->` *boolexp* `fi`.

[†]There is a deceptive similarity between this syntax and that used for implication in preconditions; this design error should be corrected in future versions.

Composition: A distributed algorithm will be represented as an interacting collection of automata. Once the automata classes necessary to construct the system to be simulated have been defined and compiled, instances of these classes need to be created so that a complete distributed system can be simulated. The number and type of each automata within the simulation is defined in a configuration file. This file also contains values with which to instantiate the class parameters for each automaton in the system.

In the formal method, it is the action names which determine whether or not two components in a system can influence one another. This carries over to Cerberus, in contrast to systems like the Software Architect's Assistant (Ng et al, 1995) where each component uses local names and explicit binding is done in a configuration language. In Cerberus, nodes which communicate synchronously can be created by having action names which contain parameters which identify the neighbors; the values of the parameters are set in a configuration file, so the instances which are intended to communicate synchronously are given matching action names.

Asynchronous communication is expressed by introducing a channel automaton between each pair of neighbours: Each node has as a parameter a set that contains the unique identifiers of all of the communication channels upon which it is to send messages. The `send` actions that place messages on the communication channels contain one of the identifiers from this set as a parameter, the other parameter is the message to be sent. Later the channel interacts synchronously with the other node in a `receive` action, where again the action in the destination node is parameterised with the identifier of the channel.

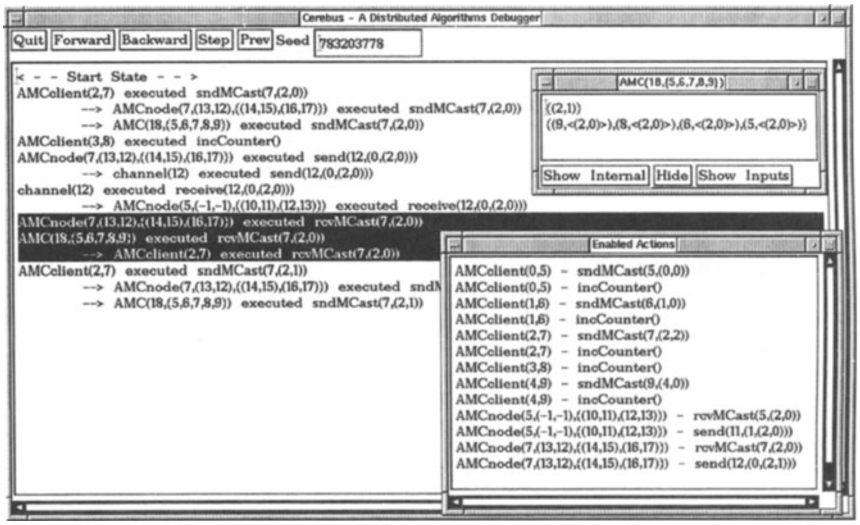
3 OVERVIEW OF THE CERBERUS TOOL

The tool includes a translator to transform the description of each automata class into a C source file. The C source file is then compiled to an object file. Once all of the automata classes required for the simulation have been compiled they are linked with the simulator and a configuration file. The configuration file specifies the number of instances of each automata class in the simulation and gives values to the parameters of the automata instances.

Specification: To specify the required (correct) behaviour of the system, an automaton that contains the same interface as the implementation must also be provided. The specification automaton models the correct behaviour of the system by only enabling those output actions that correspond to the correct results being passed to the users of the service. The specification automaton should contain the input and output class actions used by the algorithm under investigation, when it interacts with the clients. The specification automaton should be able to model the correct behaviour of the system regardless of the number of users of the service being provided. Hence, a set that contains the identifiers of each of the users of the service will almost always be a parameter for this automaton.

It is important to remember that the specification automata should contain all the actions that communicate with the client regardless of the fact that these actions may be contained in components that are physically distributed from one another. This style of specification, with a global automaton to characterize the desired service, is discussed by Fekete (1993).

Simulation: Once Cerberus is running, with linked code for the algorithm's components and for the specification, the first activity is to generate one of the possible executions



```
783203778
```

```
< - - Start State - - >
AMCclient(2,7) executed sndMCast(7,(2,0))
        --> AMCnode(7,(13,12),((14,15),(16,17))) executed sndMCast(7,(2,0))
        --> AMC(18,(5,6,7,8,9)) executed sndMCast(7,(2,0))
AMCclient(3,8) executed incCounter()
AMCnode(7,(13,12),((14,15),(16,17))) executed send(12,(0,(2,0)))
        --> channel(12) executed send(12,(0,(2,0)))
channel(12) executed receive(12,(0,(2,0)))
        --> AMCnode(5,(-1,-1),((10,11),(12,13))) executed receive(12,(0,(2,0)))
AMCnode(7,(13,12),((14,15),(16,17))) executed rcvMCast(7,(2,0))
AMC(18,(5,6,7,8,9)) executed rcvMCast(7,(2,0))
        --> AMCclient(2,7) executed rcvMCast(7,(2,0))
AMCclient(2,7) executed sndMCast(7,(2,1))
        --> AMCnode(7,(13,12),((14,15),(16,17))) executed sndM
        --> AMC(18,(5,6,7,8,9)) executed sndMCast(7,(2,1))
```

```
((2,1))
((9,<(2,0)>),(8,<(2,0)>),(6,<(2,0)>),(5,<(2,0)>))
```

```
AMCclient(0,5) - sndMCast(5,(0,0))
AMCclient(0,5) - incCounter()
AMCclient(1,6) - sndMCast(6,(1,0))
AMCclient(1,6) - incCounter()
AMCclient(2,7) - sndMCast(7,(2,2))
AMCclient(2,7) - incCounter()
AMCclient(3,8) - incCounter()
AMCclient(4,9) - sndMCast(9,(4,0))
AMCclient(4,9) - incCounter()
AMCnode(5,(-1,-1),((10,11),(12,13))) - rcvMCast(5,(2,0))
AMCnode(5,(-1,-1),((10,11),(12,13))) - send(11,(1,(2,0)))
AMCnode(7,(13,12),((14,15),(16,17))) - rcvMCast(7,(2,0))
AMCnode(7,(13,12),((14,15),(16,17))) - send(12,(0,(2,1)))
```

Figure 2 Cerberus Main Window

of the algorithm being modeled. The execution is developed step by step. At each step, the possible enabled actions are calculated. To be more precise, the system considers each action template which is an output or internal action of any component; it decides which (if any) values of the parameters in that template make both restrict clause and precondition true. Then the user may choose one of the enabled actions (they are displayed in a popup window, seen in the lower right in Figure 2); alternatively, the next action may be selected randomly from those enabled. Whichever mechanism is used, once an action is chosen, it is compared with those that the specification allows: an error reported if the action is not appropriate in the specification.

Once an action has been selected and determined not to cause a violation of the specification then the simulation can take the corresponding transition. This amounts to performing the assignments in the effect code, with the action parameters instantiated to the values that enable the action. Also, if any other components have the same action class name as input, and the chosen parameters satisfy the corresponding `restrict` section, then each of these components also executes the transition determined by its own code. The specification also takes a transition if that is appropriate.

When there are no enabled locally controlled actions within the automata that are modelling the implementation, then there are two possibilities. If the specification automaton contains enabled actions then an error is reported. This situations indicates deadlock has occurred: the specification automaton has interactions that are unfinished, however the algorithm is unable to continue. On the other hand, if the specification automaton also contains no enabled actions then a valid end state has been reached and the simulation terminates without an error. Another execution should now be simulated, until the designer is confident that no errors are present.

Execution display: When the execution of the system is stopped for any reason, be it a breakpoint or an error, details of the actions that were executed to arrive in the current state are displayed in the main window. Also, at any point where the execution is paused the state variables of each automaton in the system may be hidden or displayed. Any displayed state variables will be updated after the execution of an action. Figure 2 shows a screen shot of the main window of the system with the enabled actions window and a window containing the state variables of an automaton.

Breakpoints: When a window is opened to display the values of the state variables of an automaton, breakpoints are placed on the execution of any action within the automaton. Whenever the simulation is running freely and it chooses an action with a break point set upon it, then the simulation is paused. A description of the action and any related actions is also printed in the main window. Breakpoints are effective in both forwards and backwards traversal of the simulation and leave the system in the state after the execution of the action in both cases. Buttons on the state display window allow the breakpoints to be removed from either (or both) of the Internal and Input actions.

Execution Exploration: A set of buttons along the top of the window allow the user to control the direction of execution and the number of steps the simulation will take. The buttons `Step` and `Prev` cause the execution to take a single step either forwards or backwards. If the algorithm is to step forward then the action it executes is dependent upon whether or not the current state is the last in the sequence of states that have been visited. If it is, then the action that is executed is determined by random selection as previously explained. If the current state has been reached by executing the program in reverse then the action to be executed will be one executed when this state was first visited. If the execution of the original action in this state caused an error then no further progress in this direction is allowed.

If the direction of execution is in reverse then the values of the state variables in the state preceding the current one are restored.

The buttons `Forward` and `Backward` again execute the system in the respective direction, however instead of executing a single event they continue executing until the action executed meets a breakpoint.

4 AN EXAMPLE - ATOMIC MULTICAST

To demonstrate the way Cerberus is used, we have taken a simple algorithm from the paper by Fekete (1993). This algorithm provides the communication service called *atomic broadcast* (or sometimes also called totally ordered broadcast). This is an important building block in management of replicated data.

The algorithm can be described informally as follows. The network topology must form a tree, with each node aware of the connection to its parent. When a node receives a request from a client to broadcast a message to all other nodes, this request is firstly sent along the parent connections until it arrives at the root of the tree. The root then sends the message to all of its children and adds the message to a FIFO queue of messages to be delivered to the client. On receipt of message from its parent, each node behaves in the same fashion. This solution achieves the essential property that all clients receive the messages in the same order. it does this by allowing one node, the root of the tree, to make the decision as to the order in which messages will be received.

To show the Cerberus tool in use, a bug was added to the algorithm. To model this situation using I/O Automata, three automata classes were used. The automata class `AMCclient(node)` models the clients which are using the service. The single parameter shown for this class specifies the identifier of the protocol node with which this client communicates, to request services and receive results. Clients make requests that a message be broadcast upon the network using the Output actions `sndMCast(node, (from,msg))` where parameter `from` contains the identifier of the client that is broadcasting the message, and `msg` contains the information to be sent to each of the other clients. In this simple example, the information is simply an integer that is incremented for each message that the client sends. The `node` parameter allows the automata that are modeling the service providers to only communicate with one client.

The clients accept delivery of a message when they execute the input action `rcvMCast(node, (from,msg))`

The protocol agent process at one node which is providing the Atomic Multicast is represented by another automaton of the class `AMCnode(parent, children)` part of whose code is shown in Figure 3. The third automata class that is used in the simulation models the communication channel that provides an error free, in-order transmission of messages between neighbouring nodes in the network.

The topology of the network in which this simulation was performed has 5 nodes. The root of the tree has Client 0; its children have Client 1 and Client 2 respectively. Client 1 is at a leaf, while Client 2's node has two children with Client 3 and Client 4 respectively. Each node also has a protocol agent whose automaton identifier is 5 more than the corresponding client.

The correctness of the algorithm is specified by providing an automata that embodies the desired behaviour of the service to be provided. It contains the same interface to the clients, i.e the actions to accept a multicast request from the client and to deliver the message to the client. The possible behaviours of the specification automata are the set of actions that represent all possible sequences of send and receive events that are allowed for an atomic multicast with the topology of the implementation automata.

The specification automata maintains two state variables. The first state variable is a set, which contains messages that have been sent by a client and have not yet been delivered to any client. The second state variable is actually a collection of FIFO queues, one for each node. The messages on a queue represent those which must be delivered to the client associated with the node. When the specification automata accepts an action for multicast it is added to the set of undelivered messages. There are two possible places from which a message can be chosen for delivery. At any time the head of the queue associated with a node may be delivered to that node (and removed from the corresponding queue). If the queue of messages awaiting delivery to a client by a node is empty, then any message in the undelivered set may be chosen and be delivered by that node. The effect of executing this action is to remove the chosen message from the undelivered set, and also to append it to the FIFO queue associated with every node except the node which delivered the message with the execution of the action.

4.1 Simulating The System

Once the automata classes (representing both algorithm and specification) had been compiled and linked with the simulator, an execution was generated randomly. Very quickly an error is reported by the system. The message below is generated and the action that

```
automata AMCnode(parent : socket; children : socketSet;)
var
    i : socket;
    N : integer;
state_vars
    rcvdQueue : MsgQueue;
    pending   : NodeQueue;
initial rcvdQueue := seq(); pending := seq();
begin
    input sndMCast(node : integer; msg : MsgType;)
    restrict  node == AutomataNum;
    begin
        if
            parent.from == NONE -> begin
                forall i in children -> do
                    pending := pending + seq(QType(i.to,PktType(RECV,msg)));
                od;
                rcvdQueue := rcvdQueue + seq(msg);
            end
            true -> begin
                pending := pending + seq(QType(parent.to,PktType(SEND,msg)));
                rcvdQueue := rcvdQueue + seq(msg); /* BUG ! ! ! */
            end
        fi;
    end

    output rcvMCast(node : integer; msg : MsgType;)
    /* OMITTED */

    input receive(channel : integer; pkt : PktType;)
    restrict
        N := 0;
        (channel == parent.from) ||
        ((forall i in children -> do
            if
                channel == i.from -> N := N + 1;
            fi;
        od) && FALSE) || (N > 0);
    begin
        if
            (pkt.type == SEND) -> begin
                if
                    (parent.to != NONE) -> pending := pending + seq(QType(parent.to,pkt));
                    true -> begin
                        rcvdQueue := rcvdQueue + seq(pkt.msg);
                        forall i in children -> do
                            pending := pending + seq(QType(i.to,PktType(RECV,pkt.msg)));
                        od;
                    end
                fi;
            end
            true -> begin
                rcvdQueue := rcvdQueue + seq(pkt.msg);
                forall i in children -> do
                    pending := pending + seq(QType(i.to,pkt));
                od;
            end
        fi;
    end

    output send(channel : integer; pkt : PktType;)
    /* OMITTED */
end
```

Figure 3 Abbreviated Source for the Atomic Multicast Node

caused the automata to change to the state in which this action is enabled is printed.

```
ERROR: specification can not execute rcvMCast(7,(2,0))
```

This leads us to examine the state variables of the specification automata and of AMCnode(7). From the displays we saw that the node that tried to execute the unallowed action contained two messages to be delivered (2,0) and (2,1). Looking at the state variables of the specification automata we saw that node 7 should have been delivering the two messages (3,0) and (3,1). We also observed that the set of messages awaiting their first delivery in the specification automata includes the message (2,0) which the node in the algorithm was about to deliver. Thus the error must have been earlier, leading to message (2,0) entering the queue incorrectly.

We would now like to begin looking for the cause of the discrepencies in the state variables. The first thing we tried was to go back and look at the the delivery of any messages to the clients. We could achieve this by displaying the window that contains the node 7 state variables and choosing to not display Input actions of the specification automaton. Now the execution will pause each time the specification automata executes an Output action: these actions correspond to the delivery of messages to the clients. As the specification contains such actions for all the nodes within the network we were able to leap back through each delivery of each message to a client. On doing this we encountered only two actions that delivered messages, node 8 delivers the message (3,0) and then (3, 1).

To observe the state variables before and after the execution of an action, we use a breakpoint on that action, and then step back one statement using the button provided. In both actions of node 8, the transformation of the state variables in both the specification automata and the implementation node are as expected. The node removes the delivered message from the queue of nodes awaiting delivery and makes no other changes. The specification removes the delivered message from the set of undelivered messages and adds copy of it to each of the message queues inside the specification automata except for the queue associated with the node where the message was delivered.

The deliveries of the messages all appear to behave in the expected manner, so we decided to investigate the sending of the messages next. We did this by displaying the state of clients 2 and 3 (those that were senders of the two messages involved). Displaying these components sets breakpoints on their output actions. When each action is found in the execution, we examine the state variables of the corresponding node. When we do this for the action that transmits a message from client 2 we saw that this action adds the message (2,0) to the queue of messages to be delivered to the client, but in the specification it was not waiting to be delivered back to the client. We step through the actions executed and find that this value remains at the head of the queue until the last state before the illegal action was executed. This reveals the bug: client 2 is not the root of the tree and therefore should not add a message until it has been sent to the root of the tree first. After this part of the code has been corrected, the algorithm works as the specification says it should.

5 CONCLUSION

We have described a tool that allows the designer to experiment with a distributed algorithm in an abstract form, early in the software lifecycle. By simulating the algorithm and continuously comparing its behavior to that of a service specification, errors can be found. The key contribution of this work is that the designer can explore the inappropriate execution, moving forwards and backwards in large stages as they seek the specific aspect of the algorithm that must be fixed.

The prototype we have constructed is far from complete. In future work, we are looking to expand the flexibility of the breakpoint mechanism. At present, we set breakpoints which allow us to jump forwards or backwards to the next action or next output of a given automaton. A natural extension would be to move along the causal dependencies; thus from any point one would have a choice of moving to the next action of that automaton or else to the send (or receive) corresponding to the current recieve (or send).

Our design envisages more powerful support for exploration. We plan to allow a breakpoint to be set independently on any variable in any component's state. Thus the user could pause the execution at the next step which leads to a change in the value of that variable. In the example of section 4, as soon as we found that there was an inappropriate value in the queue in node 7, one could jump backward directly to the action that placed the incorrect value there.

Another useful facility would be the ability to give a relationship between the state of algorithm and that of specification, and use the failure of this relationship as a condition to stop the execution's simulation. Thus sort of relationship (called a "refinement" or "abstraction mapping") has been very helpful in proving protocols correct; we expect it would also be helpful in finding errors.

REFERENCES

Fekete, A. (1993), Formal Model Of Communication Services: A Case Study, *IEEE Computer* **26**(8):37–47.

Goldman, K. (1990) *Distributed Algorithm Simulation Using Input/Output Automata.* PhD Dissertation, MIT Laboratory for Computer Science.

Holzmann, G. (1991) *Design and Validation of Computer Protocols.* Prentice-Hall.

Knapp, E. (1987) Deadlock Detection in Distributed Databases, *ACM Computing Surveys*, **19**(4):303–328.

Lynch, N. and Tuttle, M. (1989) An Introduction to Input/Output Automata, *CWI-Quaterly*, **2**(3).

Mullender, S. (1993) *Distributed Systems (2nd edition).* Addison Wesley.

Ng, K., Kramer, J., Magee, J. and Dulay, N. (1995) The Software Architect's Assistant – A Visual Environment for Distributed Programming *Proceedings of 28th Hawaii International Conference on System Sciences* vol II, 254–263.

Schneider, F. (1990) Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial *ACM Computing Surveys* **22**(4):299–319.

Stevens, W. (1990) *Unix Network Programming.* Prentice Hall.

11

Debugging Parallel Programs using Temporal Logic Specifications

M. Frey
Institut für Informatik, Technische Universität München
Orleansstr. 34, D-81667 Munich, Germany, Phone: ++49 89 48095-145, Fax: ++49 89 48095-160, email: frey@informatik.tu-muenchen.de

Abstract
A new method for debugging parallel programs using temporal logic is described. The programmer can specify his hypothesis about the causes of an error in a temporal logic language. A model checking algorithm automatically checks the specification. The model is a partial order model of a class of program runs which is generated during a single run. Because it is partially ordered, errors can be detected even if they did not occur during the program run.

Keywords
Debugging, parallel and distributed programs, specification, partial order temporal logic

1 INTRODUCTION

As the number of parallel and especially distributed systems is growing, the demand for formal methods to develop parallel and distributed programs increases. Developing methods of the sequential world are often not adequate for the development of parallel and distributed systems, because of the nondeterministic nature of parallelism and the absence of global states in distributed systems. This paper investigates the debugging process after the implementation of a parallel and distributed program.

Mostly, debugging starts when the program reacts faulty in a test case. The programmer analyzes the symptoms of the error during the test case and builds a hypothesis about the cause of the error. Afterwards, he has to decide whether the hypothesis is correct or not. If the hypothesis is correct, he can eliminate the error. Otherwise, he has to build another hypothesis

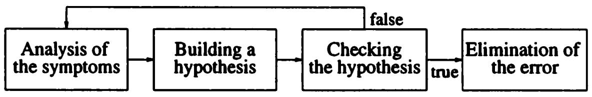


Figure 1 The process of debugging.

(see figure 1 and (Myers 1979)). Normally, the evaluation of the hypothesis is done by setting breakpoints and examining variables at breakpoints. When you have set the right breakpoints and examined the right variables, you can decide whether the hypothesis is correct or not.

Our approach supports the programmer during the evaluation of the hypothesis. He can specify the properties of the hypothesis in a temporal logic language. It is automatically checked during a second run of the test case whether the properties are satisfied or not. If the properties are satisfied, the hypothesis is not correct. Otherwise, the error is located and can be eliminated.

Our concept can be used for parallel systems which meet the shared memory paradigm, the message passing paradigm, or combinations of both. In the following we call parallel activities of a program tasks. Tasks can work on the same address space or on different address spaces including distribution. They can communicate by shared variables or by message passing. Tasks can be created dynamically during a program run (see figure 2). These model of parallel and distributed programs contain programs written in most of the existing languages or libraries.

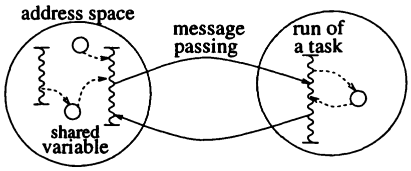


Figure 2 Model of parallel programs.

The next section gives an overview of the evaluation of the hypothesis. Section 3 introduces our model for program runs, called state action net. A description of the temporal logic is given in section 4. Section 5 describes the model checking algorithm. Finally, we give some conclusions.

2 OVERVIEW

When the programmer has specified the hypothesis, he reruns the program. During this run interesting events are recorded in a trace. Interesting events are for example task creation, communication, access to a shared variable, entering, or leaving of a function body. This trace

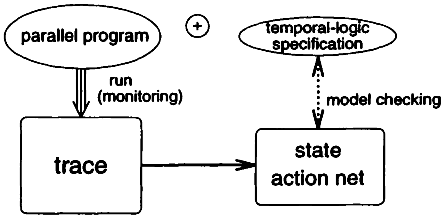


Figure 3 Debugging based on temporal-logic specifications.

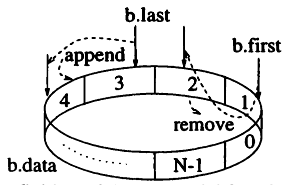
is used to generate a state action net. It contains only those dependencies between tasks of the program run which represent synchronization points given by the semantics of the program. All other dependencies between tasks of the run are arbitrary and left out. This way, a state action net does not only represent the program run by which it was generated but an equivalence class of program runs. This equivalence class can contain faulty program runs even if the original program run was correct. Finally, a model checking algorithm is applied which checks whether the temporal-logic specification is satisfied in the state action net (see figure 3).

(Hurfin, Plouzeau & Raynal 1993) present another approach for debugging parallel and distributed systems. They use atomic sequences of predicates for specification. (Garg & Waldecker 1994) use a subset of a temporal logic which allows only to specify formulas of the form

$\Box\neg(p_1 \wedge \ldots \wedge p_n)$. Both approaches can not be used for programs communicating by shared variables and programs with dynamic task creation. Their models of program runs define a partial order between events of tasks and are similar to state action nets in this way. In contrast to (Frey & Weininger 1994) this paper investigates the model checking algorithm in more detail.

Example

Figure 4 shows an example for a parallel program. This program implements a ring buffer with parallel access. The buffer is represented by the variable *b*. It contains the components *data* for the data field, *first* pointing on the element of *data* previous to the first element, and *last* pointing on the last element. The sequential function *append* inserts an element at the end of the buffer by incrementing *last* and inserting the element. The function *remove* fetches an element from the beginning of the buffer and increments *first*. This way, the buffer is implemented by a queue. The parallel function *get* contains some synchronization statements and calls *remove*. *put* also contains synchronization and calls *append*. We will use this example in the following sections.



Definition of the sequential functions remove and append

```
#define N 200
struct queue { int data[N]; int first; int last } b={{0},0,0};

put(int elem) {
  mutual exclusion w.r.t. put:
    wait until b not full;
    append(elem); }
get(int *elem) {
  mutual exclusion w.r.t. get:
    wait until b not empty;
    remove(elem); }
```

Figure 4 A ring buffer with parallel access.

3 STATE ACTION NETS

State action nets are our models of program runs. They are finite occurrence nets (cf. (Reisig 1988)) with specific kinds of nodes: actions and local states.

Actions represent executions of statements of the source code which have to take place at the same time because of the semantics of the program. For example an action modeling a synchronous communication represents two executions of statements (see figure 5). An action contains

- for each of its executions of statements the location of the statements in the source code and
- the tasks which have executed the statements.

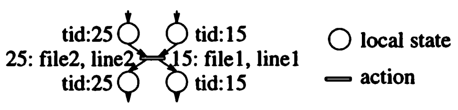


Figure 5 An action modeling a synchronous communication.

Local states contain information which is local for a task. A local state s contains

- the identification of the task ($tid(s)$) to which it belongs.
- information about the functions executed at s ($stack(s)$). It can be distinguished whether s is at the beginning of a function f (START.f), s is somewhere during the execution of f (IN.f),

or s is at the end of f (TERM.f). $stack(s)$ contains the information about the execution in the same order as the functions are called.

- a set of values for each variable var ($val(var, s)$) which is visible in the functions of $stack(s)$. A shared variable var can have more than one value in s if an action exists which is not causally ordered with s and represents a write access to var. In this case var can have in s the value before the write access as well as the value after the write access.

State action nets are occurrence nets which have to fulfill additional properties: A state action net (SAN) N is a tuple (S, A, R, I) where

1. S is a finite and nonempty set of local states.
2. A is a finite and nonempty set of actions.
3. $R \subseteq (S \times A \cup A \times S)$ is a relation with the following properties:
   (a) the transitive closure of R, R^+, is irreflexive.
   (b) for all $s \in S$:$|\ {}^\bullet s\ | \leq 1$ and $|\ s^\bullet\ | \leq 1$ where ${}^\bullet s$ is the preset and $s^\bullet$ is the postset of s.
   (c) for all $s_1, s_2 \in S$: if $s_1 \neq s_2$, $(s_1, s_2) \notin R^+$, and $(s_2, s_1) \notin R^+$ then $tid(s_1) \neq tid(s_2)$.
   (d) for all $s \in S$:$|\ {}^\bullet s\ |= 1$ and for all $a \in A$:$|\ a^\bullet\ |> 0$.
4. $I \subseteq A$ where for all $a \in I$:$|\ {}^\bullet a\ |= 0$.

The properties 1., 2., 3.(a) and 3.(b) are properties of a finite occurrence net and define that SANs are a specific kind of occurrence nets. 4. defines that I contains all initial actions. 3.(c) states that local states which belong to the same task, are not concurrent. 3.(d) defines that for each local state s an initial action a exists with $(a, s) \in R^+$.

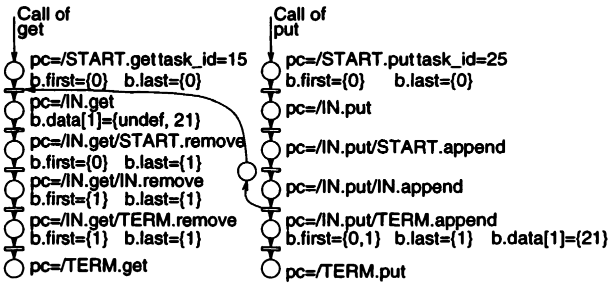


Figure 6 A state action net generated by a run of the program of figure 4.

The SAN in figure 6 is generated by a test case of the program of figure 4. In the test case *get* is called and afterwards *put* inserts 21. There is one dependency between the task executing *put* and the task executing *get* which is caused by the waiting condition of *get*.

4 TEMPORAL LOGIC SPECIFICATIONS

The temporal logic to specify the hypothesis consists of two parts:

- The local state logic (LSL) is a propositional logic to specify properties of local states. Formulas of LSL can be satisfied, unsatisfied or undefined in a local state s. They can be undefined in s if they contain variables which are not visible in the functions of $stack(s)$.
- The temporal logic (TL) is used to specify properties of the SAN and is similar to the logic in (Reisig 1988). TL contains formulas of LSL as atomic formulas. The semantics of TL-operations are defined over global states of the SAN, called slices.

A slice of an SAN (S, A, R, I) is a maximal subset $l \subseteq S$ where for all $x, y \in l$, neither $(x, y) \in R^+$ nor $(y, x) \in R^+$. A slice l' is a successor of a slice l (l' nextslice l) iff an action $a \in A$ exists with $\{s \in S | (s, a) \in R\} \subseteq l$ and $(l - \{s \in S | (s, a) \in R\}) \cup \{s \in S | (a, s) \in R\} = l'$.

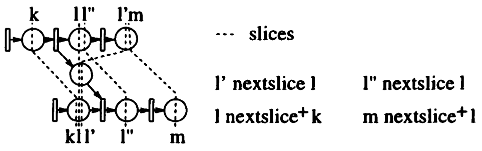


Figure 7 Slices and the relation nextslice.

Because of the properties 4. and 3.(d) of the definition of SANs in section 3 an initial slice exists for all SANs. Furthermore, it can be proven that the slice graph of an SAN which contains slices as nodes and the elements of nextslice as edges, is fully connected, directed, and acyclic.

For many interesting properties of parallel and distributed programs it is necessary to specify whether a formula is satisfied in local states of different tasks or the tasks are in the relation "is caller of". To implement many parallel and distributed systems in an efficient way, new tasks have to be created dynamically. In those system tasks can not be identified statically. So we need some kind of place holder for a task identifier, called task-id variable, and an assignment, called task-id assignment, $\mathfrak{T} : TVAR \longrightarrow TID$ where $TVAR$ is a set of variable identifiers, and TID is the set of task identifiers of a program run. The semantics of the temporal logic operations are defined by tuples $(l, \mathfrak{T})$ called cuts where l is a slice, and $\mathfrak{T}$ is a task-id assignment.

The logic has some differences to other logics like LTL or CTL* (cf. (Emerson 1990)):

- The model for program runs are sequences in LTL and trees in CTL*. The slice graph defines a partial order between slices. This partial order can of course be unwounded to a tree or a sequence of slices and the temporal logic operations, except **until**, can be expressed by CTL*.
- In contrast to LTL and CTL* our logic has the following compositional property: If a formula is satisfied in an SAN $N_1 = (S_1, A_1, F_1, I_1)$ then it is also satisfied in an SAN $N_2 = (S_2, A_2, F_2, I_2)$ where $S_1 \subseteq S_2$, $A_1 \subseteq A_2$, $F_2 |_{S_1 \times A_1 \cup A_1 \times S_1} = F_1$, and the functions, task-ids, and variables of $s \in S_1$ are different to those of $s \in S_2 - S_1$ (see (Reisig 1988) for details). Because of this property, it is possible to test and debug modules separately. After composition of the modules a further test of the properties of the modules is not necessary.

The syntax of LSL contains predicates and formulas. A predicate may be *p_expr*, **start.***f*, **in.***f*, or **term.***f* where *f* is a function, and *p_expr* is a side effect free expression of the programming language which returns a boolean value, and may contain program variables. A formula of LSL (LSL-formula) may be a predicate, (p **or** q), (p **and** q), or **not** (p) where p and q are formulas.

The semantics of LSL is defined over local states. A predicate or a formula p of LSL can be satisfied in a local state s ($s \models p$), unsatisfied in s ($s \not\models p$), or undefined in s ($s \mid\!\uplus p$):

- $s \models$ *p_expr* iff for all values of each program variable, *p_expr* evaluates to true.
  $s \not\models$ *p_expr* iff a value of each program variable exist where *p_expr* evaluates to false.
  $s \mid\!\uplus$ *p_expr* iff a program variable of *p_expr* is not visible in the functions of $stack(s)$.
- $s \models$ **start.***f* iff START.*f* $\in stack(s)$, $s \not\models$ **start.***f* else.
- $s \models$ **in.***f* iff START.*f* $\in stack(s)$, IN.*f* $\in stack(s)$, or TERM.*f* $\in stack(s)$, $s \not\models$ **in.***f* else.
- $s \models$ **term.***f* iff TERM.*f* $\in stack(s)$, $s \not\models$ **term.***f* else.
- $s \models$ (p **and** q) iff $s \models p$ and $s \models q$.
  $s \not\models$ (p **and** q) iff [$s \not\models p$ and [$s \not\models q$ or $s \models q$]] or [$s \models p$ and $s \not\models q$], $s \mid\!\uplus$ (p **and** q) else.
- $s \models$ (p **or** q) iff [$s \models p$ and [$s \not\models q$ or $s \models q$]] or [$s \not\models p$ and $s \models q$].
  $s \not\models$ (p **or** q) iff $s \not\models p$ and $s \not\models q$, $s \mid\!\uplus$ (p **or** q) else.

- $s \models$ **not** (p) iff $s \not\models p$. $s \not\models$ **not** (p) iff $s \models p$, and $s \mathrel{|\!\!\ominus}$ **not** (p) else.

An expression of TL (TL-expression) may be *tvar*:(*lsf*), **not** (p), (p **and** q), (p **or** q), **next** (p), **sometime** (p), **always** (p), (p **before** q), or (p **until** q) where *tvar* is a task-id variable, *lsf* is a LSL-formula, and p, q are TL-expressions. A formula of TL (TL-formula) may be *expr*, **forall** *tvar* (p), **exists** *tvar* (p), (p **if** *tvar* **==** *tvar'*), (p **if** *tvar* **!=** *tvar'*), or (p **if** *tvar* **== caller(***tvar'***)**) where p is a TL-formula, *expr* is a TL-expression and *tvar*, *tvar'* are task-id variables.

The semantics of TL is based on cuts. A formula p of TL can be satisfied in a cut $\mathfrak{C} = (l, \mathfrak{T})$ ($\mathfrak{C} \models p$), unsatisfied in $\mathfrak{C}$ ($\mathfrak{C} \not\models p$), or undefined in $\mathfrak{C}$ ($\mathfrak{C} \mathrel{|\!\!\ominus} p$):

- $\mathfrak{C} \models$ *tvar*:(*lsf*) iff an $s \in l$ exists where $\mathfrak{T}(\mathit{tvar}) = \mathit{tid}(s)$, and $s \models \mathit{lsf}$.
  $\mathfrak{C} \not\models$ *tvar*:(*lsf*) iff an $s \in l$ exists where $\mathfrak{T}(\mathit{tvar}) = \mathit{tid}(s)$, and $s \not\models \mathit{lsf}$. $\mathfrak{C} \mathrel{|\!\!\ominus}$ *tvar*:(*lsf*) else.
- $\mathfrak{C} \models$ **not** (p) iff $\mathfrak{C} \not\models p$. $\mathfrak{C} \not\models$ **not** (p) iff $\mathfrak{C} \models p$, and $\mathfrak{C} \mathrel{|\!\!\ominus}$ **not** (p) else.
- $\mathfrak{C} \models$ (p **and** q) iff $\mathfrak{C} \models p$ and $\mathfrak{C} \models q$. $\mathfrak{C} \not\models$ (p **and** q) iff $\mathfrak{C} \not\models p$ or [$\mathfrak{C} \models p$ and $\mathfrak{C} \not\models q$].
  $\mathfrak{C} \mathrel{|\!\!\ominus}$ (p **and** q) else.
- $\mathfrak{C} \models$ (p **or** q) iff $\mathfrak{C} \models p$ or [$\mathfrak{C} \not\models p$ and $\mathfrak{C} \models q$]. $\mathfrak{C} \not\models$ (p **or** q) iff $\mathfrak{C} \not\models p$ and $\mathfrak{C} \not\models q$,
  $\mathfrak{C} \mathrel{|\!\!\ominus}$ (p **or** q) else.
- $\mathfrak{C} \models$ **next** (p) iff a slice l' exists where l' nextslice l and $(l', \mathfrak{T}) \models p$.
  $\mathfrak{C} \not\models$ **next** (p) iff for all slices l' where l' nextslice l, $(l', \mathfrak{T}) \not\models p$ or $(l', \mathfrak{T}) \mathrel{|\!\!\ominus} p$.
- $\mathfrak{C} \models$ **sometime** (p) iff a slice l' exists where l' nextslice^{+} l and $(l', \mathfrak{T}) \models p$.
  $\mathfrak{C} \not\models$ **sometime** (p) iff for all slices l' where l' nextslice^{+} l, $(l', \mathfrak{T}) \not\models p$ or $(l', \mathfrak{T}) \mathrel{|\!\!\ominus} p$.
- $\mathfrak{C} \models$ **always** (p) iff for all slices l' where l' nextslice^{+} l, $(l', \mathfrak{T}) \models p$ or $(l', \mathfrak{T}) \mathrel{|\!\!\ominus} p$.
  $\mathfrak{C} \not\models$ **always** (p) iff a slice l' exists where l' nextslice^{+} l and $(l', \mathfrak{T}) \not\models p$.
- $\mathfrak{C} \models$ (p **before** q) iff for all slices l' where [l' nextslice^{+} l and $(l', \mathfrak{T}) \models q$], a slice l'' exists where [l'' nextslice^{+} l, and l' nextslice^{+} l''], and $(l'', \mathfrak{T}) \models p$.
  $\mathfrak{C} \not\models$ (p **before** q) iff a slice l' exists where [l' nextslice^{+} l, and $(l', \mathfrak{T}) \models q$] and for all slices l'' where [l'' nextslice^{+} l and l' nextslice^{+} l''], $(l'', \mathfrak{T}) \not\models p$ or $(l'', \mathfrak{T}) \mathrel{|\!\!\ominus} p$.
- $\mathfrak{C} \models$ (p **until** q) iff a slice l' exists where [l' nextslice^{+} l, and $(l', \mathfrak{T}) \models q$], and for all slices l'' where [l'' nextslice^{+} l and l' nextslice^{+} l''], $(l'', \mathfrak{T}) \models p$, or $(l'', \mathfrak{T}) \mathrel{|\!\!\ominus} p$.
  $\mathfrak{C} \not\models$ (p **until** q) iff for all slices l' where [l' nextslice^{+} l and $(l', \mathfrak{T}) \models q$], a slice l'' exists where [l'' nextslice^{+} l, and l' nextslice^{+} l''], and $(l'', \mathfrak{T}) \not\models p$.
- $\mathfrak{C} \models$ **forall** *tvar* (p) iff $(l, \mathfrak{T}\frac{\mathit{tvar}}{t}) \models p$* or $(l, \mathfrak{T}\frac{\mathit{tvar}}{t}) \mathrel{|\!\!\ominus} p$ for all $t \in TID$.
  $\mathfrak{C} \not\models$ **forall** *tvar* (p) iff $(l, \mathfrak{T}\frac{\mathit{tvar}}{t}) \not\models p$ for some $t \in TID$.
- $\mathfrak{C} \models$ **exists** *tvar* (p) iff $(l, \mathfrak{T}\frac{\mathit{tvar}}{t}) \models p$ for some $t \in TID$.
  $\mathfrak{C} \not\models$ **exists** *tvar* (p) iff $(l, \mathfrak{T}\frac{\mathit{tvar}}{t}) \not\models p$ or $(l, \mathfrak{T}\frac{\mathit{tvar}}{t}) \mathrel{|\!\!\ominus} p$ for all $t \in TID$.
- $\mathfrak{C} \models$ (p **if** *tvar op tvar'*) iff $\mathfrak{C} \models p$ and $\mathfrak{T}(\mathit{tvar})\mathrm{op}\mathfrak{T}(\mathit{tvar'})$. $\mathfrak{C} \not\models$ (p **if** *tvar op tvar'*) iff $\mathfrak{C} \not\models p$ and $\mathfrak{T}(\mathit{tvar})\mathrm{op}\mathfrak{T}(\mathit{tvar'})$. $\mathfrak{C} \mathrel{|\!\!\ominus}$ (p **if** *tvar op tvar'*) else. $op \in \{==, !=\}$.
- $\mathfrak{C} \models$ (p **if** *tvar* **== caller(***tvar'***)**) iff $\mathfrak{C} \models p$ and task $\mathfrak{T}(\mathit{tvar})$ is the task which has called task $\mathfrak{T}(\mathit{tvar'})$. $\mathfrak{C} \not\models$ (p **if** *tvar* **== caller(***tvar'***)**) iff $\mathfrak{C} \not\models p$ and task $\mathfrak{T}(\mathit{tvar})$ is the task which has called task $\mathfrak{T}(\mathit{tvar'})$. $\mathfrak{C} \mathrel{|\!\!\ominus}$ (p **if** *tvar* **== caller(***tvar'***)**) else.

A TL-formula p is satisfied in an SAN N iff for all slices l of N and all task-id assignments $\mathfrak{T}$ it is valid that $(l, \mathfrak{T}) \models p$ or $(l, \mathfrak{T}) \mathrel{|\!\!\ominus} p$. A TL-formula p is unsatisfied in an SAN N iff a slice l of N and a task-id assignment $\mathfrak{T}$ exist where $(l, \mathfrak{T}) \not\models p$.

We give an example by specifying some properties of the program of figure 4. A necessary property of the program is mutual exclusion between tasks executing *append*:

not(t1:(in.append) and t2:(in.append)) if t1 != t2

Another property of the program is mutual exclusion of *remove* and *append* when both are

*$\mathfrak{T}\frac{\mathit{tvar}}{t}(\mathit{tvar'}) = \mathfrak{T}(\mathit{tvar'})$ iff $\mathit{tvar'} \neq \mathit{tvar}$; $\mathfrak{T}\frac{\mathit{tvar}}{t}(\mathit{tvar}) = t$.

working on the same element of *data*. This is the case, when at most one element is in the buffer:

not(t1:(in.append and (b.last-b.first)%N<=1) and t2:(in.remove and (b.last-b.first)%N<=1))

5 DETECTING DEVIATIONS

This section describes a model checking algorithm which detects deviations between a specification and an SAN. Generally, model checking methods can be divided into two classes: Tableau based methods (cf. (Winskel 1991)) and global methods (cf. (Clarke, Grumberg & Long 1993)). Our model checking method is a global method, because subformulas of a formula are marked with cuts. It differs from other global methods because it is working on a model of program runs and the model have not to be unfolded. Another difference to tableau based methods as well as global methods is that not all cuts have to be calculated. Only those cuts must be calculated which are necessary to decide whether a TL-formula is satisfied. The model checking algorithm can be divided into the following steps:

1. Negation of TL-formulas
2. Transformation of negated TL-formulas
3. Construction of formula trees
4. Checking local state formulas
5. Checking TL-formulas

5.1 Negation of TL-formulas

Given a TL-formula p and an SAN N, we have to check whether p is satisfied in N. If p is unsatisfied, the programmer wants to know why p is unsatisfied. He is interested in the cuts and local states of N in which p and subformulas of p are unsatisfied. Therefore, we have to calculate all cuts in which **not** p is satisfied.

5.2 Transformation of TL-formulas

To check whether a TL-formula is satisfied, we have to find a set of TL-operations where a logically equivalent formula q containing only these operations exists for every TL-formula p. We require that q can be more efficiently checked than p.

The set of TL-operations which we use contains **and, or, next, all_next, always, sometime, until**, and **before**. It contains the additional operation **all_next** and does not contain the negation for the following reason: If we want to detect all cuts of N which unsatisfy a TL-formula p and satisfy **not** p, we have to detect all cuts in which p is satisfied and all cuts in which p is undefined, because p is unsatisfied in all cuts of N without the cuts in which p is satisfied or undefined. A more efficient method is to transform **not** p into a logically equivalent TL-formula which contains no negation in the temporal-logic part. Before we introduce the logical equivalences, we define the semantics of the additional operation:

$(l,\mathfrak{T}) \models$ **all_next** (p) iff for all slices l' where l' nextslice l $(l',\mathfrak{T}) \models p$ or $(l',\mathfrak{T}) \mid\!\uplus\ p$.
$(l,\mathfrak{T}) \not\models$ **all_next** (p) iff a slice l' exists with l' nextslice l and $(l',\mathfrak{T}) \not\models p$.

We have to transform the TL-formulas **not** p, into logically equivalent formulas which do not contain the operation **not**. For these transformations we use the following equivalences:

not(p **and** q**)** is equivalent to **(not** p **or not** q**)**; **not next** p is equivalent to **all_next not** p;
not sometime p is equivalent to **always not** p; **not(**p **until** q**)** is equivalent to **(not** p **before** q**)**;

not *tvar*:(*lsf*) is equivalent to *tvar*:(**not** *lsf*); **not not** *p* is equivalent to *p*;
not forall *tvar*(*p*) is equivalent to **exists** *tvar*(**not** *p*);
not (*p* **if** *ifpart*) is equivalent to ((**not** *p*) **if** *ifpart*);

5.3 Construction of formula trees

Formula trees are used to evaluate the TL-formula. They are the basic structure of the model checking concept. A formula tree can contain two different kinds of nodes:

- Nodes containing cuts which satisfy a TL-formula *p* (s-node(*p*)).
- Nodes containing cuts which leave a TL-formula *p* undefined (u-node(*p*)).

A formula tree of a TL-formula is similar to a syntax tree, but there are some differences. The formula tree of a formula *q′* is defined as follows:

- s-node(*tvar*:(*lsf*)) and u-node(*tvar*:(*lsf*)) have no successor nodes.
- The successor nodes of s-node((*p* **and** *q*)) are s-node(*p*) and s-node(*q*).
  The successor nodes of u-node((*p* **and** *q*)) are u-node(*p*), u-node(*q*), and s-node(*p*).
- The successor nodes of s-node((*p* **or** *q*)) are s-node(*p*), s-node(*q*), and s-node(**not** *p*).
  The successor nodes of u-node((*p* **or** *q*)) are u-node(*p*), u-node(*q*), and s-node(**not** *p*).
- s-node(**next** *p*), s-node(**exists** *tvar*(*p*)), and s-node((*p* **if** *ifpart*)) have the successor s-node(*p*).
- The successor nodes of s-node(**all_next** *p*) are s-node(*p*) and u-node(*p*).
- The successor node of s-node(**sometime** *p*) is s-node(*p*).
- The successor node of s-node(**always** *p*) and s-node(**forall** *tvar*(*p*)) is s-node(**not** *p*).
- The successor nodes of s-node((*p* **until** *q*)) are s-node(**not** *p*) and s-node(*q*).
- The successor nodes of s-node((*p* **before** *q*)) are s-node(*p*) and s-node(*q*).
- The root of the formula tree of *q′* is the s-node(*q′*).

The u-nodes of the TL-expressions **next** *p*, **all_next** *p*, **sometime** *p*, **always** *p*, (*p* **until** *q*), and (*p* **before** *q*) contain no cuts and no successor nodes, because these TL-expressions are never undefined. The u-nodes are only necessary for the calculation of cuts for TL-expressions of the form **all_next** *p*. TL-formulas may not be subformulas of TL-expressions and u-nodes are not necessary for them. A node of a formula **not** *p* is the node of the transformed formula using the transformations of section 5.2.

Figure 8 shows the formula tree of the formula
t1:(term.append) implies (t2:(b.data[1]==21) until t3:(start.remove))
after the negation and transformation.

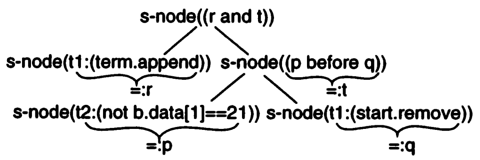


Figure 8 An example for a formula tree.

5.4 Checking local state formulas

After the construction of the formula tree cuts are calculated for the leaf nodes.

To calculate the cuts for s-node(r) where *r*=*tvar*:(*lsf*), we have to run through the SAN and check for each local state s if *lsf* is satisfied in s. *lsf* contains expressions $e_1 \ldots e_n$ and e_i contains program variables $var\_i_1 \ldots var\_i_m$. Cuts satisfying r are calculated as follows:

```
if $var\_1_1 \ldots var\_n_m$ are visible in $stack(s)$
then $C_1 := true$;
     for all $v\_1 \in val(var\_1_1, s)$ do ... for all $v\_1_m \in val(var\_1_m, s)$ do
        if $e_1 \frac{var\_1_1 \ldots var\_1_m}{v\_1 \ \ldots v\_1_m} = false$† then $C_1 := false$
        ⋮
     $C_n := true$;
     for all $v\_1 \in val(var\_n_1, s)$ do ... for all $v\_n_m \in val(var\_n_m, s)$ do
        if $e_n \frac{var\_n_1 \ldots var\_n_m}{v\_1 \ \ldots v\_n_m} = false$ then $C_n := false$
     if $lsf \frac{e_1 \ldots e_n}{C_1 \ldots C_n}$ then $\mathfrak{T}(tvar) := \{tid(s)\}$;
                        $R := slices(s)$;
                        forall $l \in R$ do   s-node($r$):=s-node($r$) $\cup$ $(l, \mathfrak{T})$
```

The function $slices(s)$ returns all slices which contain the local state s.

The cuts for u-node(r) are calculated as follows:

```
if $var\_1_1$ is not visible in $stack(s)$ or ... or $var\_n_m$ is not visible in $stack(s)$
then $\mathfrak{T}(tvar) := \{tid(s)\}$;
     $R := slices(s)$;
     forall $l \in R$ do   u-node($r$):=u-node($r$)$\cup$ $(l, \mathfrak{T})$
```

The complexity for calculating the cuts in the leave node is O(B*C) where B is the number of LSL-formulas in the formula, and C is the number of cuts of an SAN.

5.5 Checking TL-formulas

This section describes how cuts of the inner nodes of the formula tree are calculated. Beginning with cuts of the leave nodes, we calculate the cuts of the other nodes in a single bottom-up run through the formula tree. We calculate the cuts of an inner node of the formula tree using the cuts of its successor nodes and a function which depends only on the TL-operation and the kind (s-node or u-node) of the node:

The cuts of an s-node((p **and** q)) are calculated by the following algorithm:

```
for all $(l, \mathfrak{T}) \in$ s-node($p$) do  for all $(l', \mathfrak{T}') \in$ s-node($q$) do
   if $l = l'$ and $comp(\mathfrak{T}, \mathfrak{T}', VAR(p) \cap VAR(q))$
   then $\mathfrak{T}'' = merge(\mathfrak{T}, VAR(p), \mathfrak{T}', VAR(q))$;
        s-node(($p$ and $q$)):=s-node(($p$ and $q$)) $\cup(l, \mathfrak{T}'')$;
```

$comp(\mathfrak{T}, \mathfrak{T}', VAR)$ checks whether $\mathfrak{T}(v) = \mathfrak{T}'(v)$ for all $v \in VAR$. $VAR(p)$ is the set of all task-id variables of p. The complexity of $comp(\mathfrak{T}, \mathfrak{T}', VAR)$ is O(H) where H=$|\ VAR\ |$. $merge(\mathfrak{T}, VAR(p), \mathfrak{T}', VAR(q))$ calculates a task-id assignment $\mathfrak{T}''$ with $\mathfrak{T}''(v) = \mathfrak{T}(v)$ for all $v \in VAR(p)$ and $\mathfrak{T}''(v) = \mathfrak{T}'(v)$ for all $v \in VAR(q)$. The complexity is O(H) where H=$|\ VAR(p) \cup VAR(q)\ |$. The complexity of the algorithm is O(C^2*H) where C is the number of cuts of an SAN.

The cuts of an u-node((p **and** q)) are calculated as follows:

† When $p\frac{x_1 \ldots x_n}{y_1 \ldots y_n}$ is evaluated, the terms x_i of p are simultaneously substituted by y_i for all $i = 1 \ldots n$.

```
for all $(l, \mathfrak{T}) \in$ s-node($p$) do  for all $(l', \mathfrak{T}') \in$ u-node($q$) do
  if $l = l'$ and $comp(\mathfrak{T}, \mathfrak{T}', VAR(p) \cap VAR(q))$
  then ⌈ $\mathfrak{T}'' = merge(\mathfrak{T}, VAR(p), \mathfrak{T}', VAR(q))$;
       ⌊ u-node(($p$ and $q$)):=u-node(($p$ and $q$)) $\cup (l, \mathfrak{T}'')$;
for all $(l, \mathfrak{T}) \in$ u-node($p$) do  for all $\mathfrak{T}'$ do
  if $comp(\mathfrak{T}', \mathfrak{T}, VAR(p))$ then u-node(($p$ and $q$)):=u-node(($p$ and $q$)) $\cup (l, \mathfrak{T}')$;
```

The algorithm has O(C^2*H) complexity.

The following algorithm calculates the cuts of s-node((*p* **or** *q*)):

```
for all $(l, \mathfrak{T}) \in$ s-node($q$) do  for all $(l', \mathfrak{T}') \in$ s-node(not $p$) do
  if $l = l'$ and $comp(\mathfrak{T}, \mathfrak{T}', VAR(p) \cap VAR(q))$
  then ⌈ $\mathfrak{T}'' = merge(\mathfrak{T}, VAR(p), \mathfrak{T}', VAR(q))$;
       ⌊ s-node(($p$ or $q$)):=s-node(($p$ or $q$)) $\cup (l, \mathfrak{T}'')$;
for all $(l, \mathfrak{T}) \in$ s-node($p$) do  for all $\mathfrak{T}'$ do
  if $comp(\mathfrak{T}', \mathfrak{T}, VAR(p))$ then s-node(($p$ or $q$)):=s-node(($p$ or $q$)) $\cup (l, \mathfrak{T}')$;
```

The algorithm is also of O(C^2*H).

The cuts of an u-node((*p* **or** *q*)) are calculated using the cuts of the successor nodes:

```
for all $(l, \mathfrak{T}) \in$ s-node(not $p$) do  for all $(l', \mathfrak{T}') \in$ u-node($q$) do
  if $l = l'$ and $comp(\mathfrak{T}, \mathfrak{T}', VAR(p) \cap VAR(q))$
  then ⌈ $\mathfrak{T}'' = merge(\mathfrak{T}, VAR(p), \mathfrak{T}', VAR(q))$;
       ⌊ u-node(($p$ or $q$)):=u-node(($p$ or $q$)) $\cup (l, \mathfrak{T}'')$;
for all $(l, \mathfrak{T}) \in$ u-node($p$) do  for all $\mathfrak{T}'$ do
  if $comp(\mathfrak{T}', \mathfrak{T}, VAR(p))$ then u-node(($p$ or $q$)):=u-node(($p$ or $q$)) $\cup (l, \mathfrak{T}')$;
```

The complexity is O(C^2*H).

The cuts of a node s-node(**next** *p*) are calculated as follows:

```
for all $(l, \mathfrak{T}) \in$ s-node($p$) do ⌈ $R := PRED((l, \mathfrak{T}))$;
                                    ⌊ s-node(next $p$):=s-node(next $p$) $\cup R$;
```

$PRED((l, \mathfrak{T}))$ calculates all cuts $(l', \mathfrak{T})$ with l nextslice l'. The complexity of $PRED$ is O(K) where K is the maximal number of local states which are concurrent. The algorithm for calculating the cuts of s-node(**next** *p*) has O(C*K) complexity.

The following algorithm calculates the cuts of s-node(**all_next** *p*):

```
$B :=$ s-node($p$) $\cup$ u-node($p$);
for all $(l, \mathfrak{T}) \in B$ do  for all $(l', \mathfrak{T}) \in PRED((l, \mathfrak{T}))$ do
  ⌈ $C := true$;
    for all $(l'', \mathfrak{T}) \in SUCC((l', \mathfrak{T}))$ do
      ⌈ $C1 := false$;
        for all $(\hat{l}, \hat{\mathfrak{T}}) \in B$ do   if $l'' = \hat{l}$ and $comp(\mathfrak{T}, \hat{\mathfrak{T}}, VAR(p))$ then $C1 := true$;
      ⌊ if not $C1$ then $C := false$;
  ⌊ if $C$ then s-node(all_next $p$):= s-node(all_next $p$) $\cup (l', \mathfrak{T})$;
```

$SUCC((l, \mathfrak{T}))$ calculates all cuts $(l', \mathfrak{T})$ with l' nextslice l. The complexity of $SUCC$ is O(K). The complexity of calculating the cuts of s-node(**all_next** *p*) is O(C^2*K^2*H).

The cuts of s-node(**sometime** *p*) are calculated as follows:

```
for all $(l, \mathfrak{T}) \in$ s-node($p$) do  for all $l'$ do
  if $l$ nextslice$^+$ $l'$ then s-node(sometime $p$):=s-node(sometime $p$) $\cup$ $(l', \mathfrak{T})$;
```

The relation nextslice$^+$ can be checked in O(K^2). The algorithm is of O(C*S*K^2) where S is the number of slices of an SAN.

The cuts of an s-node(**always** p) are calculated in the following manner:

```
for all $(l, \mathfrak{T})$ do ⌜$C := true$;
                    for all $(l', \mathfrak{T}') \in$ s-node(not $p$) do
                        if $comp(\mathfrak{T}, \mathfrak{T}', VAR(p))$ and $l'$ nextslice$^+$ $l$ then $C := false$;
                   ⌞if $C$ then s-node(always $p$):=s-node(always $p$) $\cup$ $(l, \mathfrak{T})$;
```

The algorithm is of O(C^2*(H+K^2)).

The following algorithm describes how the cuts of s-node((p **until** q)) are calculated:

```
for all $(l, \mathfrak{T}) \in$ s-node($q$) do   for all (l',$\mathfrak{T}'$) do
    if $comp(\mathfrak{T}, \mathfrak{T}', VAR(q))$ and $l$ nextslice$^+$ $l'$
    then ⌜$C := true$;
           for all $(l'', \mathfrak{T}'') \in$ s-node(not $p$) do
               if $comp(\mathfrak{T}', \mathfrak{T}'', VAR(p))$, $l''$ nextslice$^+$ $l'$ and $l$ nextslice$^+$ $l''$ then $C := false$;
          ⌞if $C$ then s-node(($p$ until $q$)):=s-node(($p$ until $q$)) $\cup(l', \mathfrak{T}')$;
```

The complexity is O(C^3*(H+K^2)).

The cuts of an s-node((p **before** q)) are calculated as follows:

```
for all $(\hat{l}, \hat{\mathfrak{T}})$ do
   ⌜$C := true$;
    for all $(l, \mathfrak{T}) \in$ s-node($q$) do
       if $comp(\mathfrak{T}, \hat{\mathfrak{T}}, VAR(q))$, and $l$ nextslice$^+$ $\hat{l}$
       then⌜$C1 := true$;
             for all $(l'', \mathfrak{T}'') \in$ s-node($q$) do
                if $comp(\mathfrak{T}'', \hat{\mathfrak{T}}, VAR(q))$, $l$ nextslice$^+$ $l''$, and $l''$ nextslice$^+$ $\hat{l}$ then $C1 := false$;
             if $C1$
             then⌜$C2 := true$;
                   for all $(l', \mathfrak{T}') \in$ s-node($p$) do
                      if $comp(\mathfrak{T}', \hat{\mathfrak{T}}, VAR(p))$, $l'$nextslice$^+$ $\hat{l}$, and $l$nextslice$^+$ $l'$then$C2 := false$;
                  ⌞if $C2$ then $C := false$;
            ⌞
   ⌞if $C$ then s-node(($p$ before $q$)):=s-node(($p$ before $q$)) $\cup$ $(\hat{l}, \hat{\mathfrak{T}})$;
```

The complexity is O(C^3*(H+K^2)).

The following algorithm calculates the cuts of s-node(**forall** *tvar*(p)):

```
for all $(l, \mathfrak{T})$ do ⌜$C := true$;
                    for all $t \in TID$ do  for all $(l', \mathfrak{T}') \in$ s-node(not $p$) do
                        if $l = l'$ and $comp(\mathfrak{T}\frac{t}{tvar}, \mathfrak{T}', VAR(p))$ then $C := false$;
                   ⌞if $C$ then s-node(forall tvar($p$)):=s-node(forall tvar($p$)) $\cup$ $(l, \mathfrak{T})$;
```

The algorithm is of O(C^2*Z*H) where Z=| TID |.

An s-node(**exists** *tvar*(p)) contains the same cuts as its successor node.

The cuts of s-node((p **if** *ifpart*)) are calculated as follows:

```
for all $(l, \mathfrak{T}) \in$ s-node($p$) do
    if $ifpart(\mathfrak{T})$ then s-node(($p$ if ifpart)):=s-node(($p$ if ifpart)) $\cup$ $(l, \mathfrak{T})$;
```

The algorithm is of O(C).

Using these algorithms for calculating the cuts of the inner nodes, the formula tree is evaluated bottom-up. If a slice is calculated for the root node of a formula tree, the specification is unsatisfied. The result of the checking algorithm states whether the specification is satisfied or not. If the specification is not satisfied the formulas which are not satisfied and a subnet of the state action net can be obtained to locate the error. The complexity of the whole algorithm is $O(L*C^3*(H+K^2))$ where L is the number of operations of the formula.

6 SUMMARY

This paper introduces a new method for debugging parallel programs using temporal logic. The method is based on model checking where in contrary to other model checking methods models of program runs are used. The main advantages of the method are that complex global properties can be specified and checked, errors can be detected even if they do not occur during the program run, and systems with dynamic creation of processes can be investigated. Our further work will include investigations to extend the method to use it for testing parallel programs.

7 REFERENCES

Clarke, E., Grumberg, O. & Long, D. (1993), Verification tools for finite-state concurrent systems, *in* J. de Bakker, W.-P. de Roever & G. Rozenberg, eds, 'A Decade of Concurrency, REX School/Symposium, Noordwijkerhout, The Netherlands', LNCS 457, Springer, Berlin, pp. 840–851.

Emerson, E. (1990), Temporal and modal logic, *in* J. van Leeuwen, ed., 'Handbook of Theoretical Computer Science', Vol. B, Elsevier Science Publishers, Amsterdam, pp. 996–1072.

Frey, M. & Weininger, A. (1994), A temporal logic language for debugging parallel programs, *in* 'Proceedings of the 20th EUROMICRO Conference, Liverpool, England', Euromicro, IEEE, pp. 170–178.

Garg, V. & Waldecker, B. (1994), 'Detection of weak unstable predicates in distributed systems', *IEEE Trans. on Parallel and Distributed Systems* **5**(3), 229–307.

Hurfin, M., Plouzeau, N. & Raynal, M. (1993), Detecting atomic sequences of predicates in distributed computations, *in* 'ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA', ACM, ACM Press, pp. 32–42.

Myers, G. (1979), *The Art of Software Testing*, Wiley, New York.

Reisig, W. (1988), Temporal logic and causality in concurrent systems, *in* 'Concurrency 88', LNCS 335, Springer, Berlin, pp. 121–139.

Winskel, G. (1991), 'A note on model checking the modal μ-calculus', *Theoretical Computer Science*. **83**(1), 157–167.

8 BIOGRAPHY

Maximilian Frey studied Computer Science at Technische Universität München. He received his MS degree from Technische Universität München in Computer Science (Diplom-Informatiker UNIV.) in 1993. He is currently a PhD student of Computer Science at Technische Universität München. His research interests include specification, testing and performance analysis of parallel and distributed systems.

12
OPERA : A Toolbox For Loop Parallelization

Vincent Loechner and Catherine Mongenet
Université Louis Pasteur de Strasbourg, Laboratoire ICPS
Pôle API, Boulevard Sébastien Brant, 67400 Illkirch, France
Phone : (33) 88 65 50 37. Fax : (33) 88 65 50 61
email : {loechner,mongenet}@icps.u-strasbg.fr

Abstract
This paper presents the mathematical notions for the parallelization of DO-Loops used in the tool OPERA currently under development in our team. It aims at giving the user an environment to parallelize problems described by systems of parameterized affine recurrence equations which formalize single-assignment loop nests. The parallelization technique used in OPERA is based on a classical linear space×time transformation. Its objectives are to visualize the affine dependences of a problem and to propose a set of different parallel solutions depending on various architectural constraints.

Keywords
loop parallelization, parameterized affine recurrence equations, parameterized domains.

1 INTRODUCTION

Scientific computing as well as many other application domains, has always required large amounts of memory and hours of CPU time. An answer to achieve such high-performance computing can be found in the use of parallelism. Over the past decades many improvements have been made in the evolution of parallel machines, parallel languages and parallel environments.

Scientific programs are characterized by the well-known *90-10 rule* : 90% of the CPU time is spent in 10% of the code, that is to say the DO-Loops. Our objective is to develop techniques and tools to efficiently and automatically parallelize such loops and to show how various architectural constraints can be taken into account in order to synthesize different parallel solutions. These parallelization techniques are founded on sound and formal mathematical notions in order to guaranty the correctness of the synthesized parallel solutions and offer a good framework for their performance evaluation. Our work is based on the use of Systems of Recurrence Equations which are a formal description of single assignment DO-Loops. It finds its foundations on the many works on systolic array synthesis and their latest extensions to deal with affine recurrences. These recurrence equations are defined over convex integer polyhedra. In order to find a parallel solution, we apply on such polyhedra a linear space×time transformation. Therefore our formal methodology is based on the definition of such polyhedra and their manipulation using linear algebra.

This paper presents the tool OPERA* currently under development in our team. This prototype aims at giving the user an environment to express and solve problems defined by systems of parameterized affine recurrence equations (PARE in the following). Parameterized equations formalize loop nests with loop bounds depending on parameters, i.e. whose values are unknown at compile time. One of the interests of OPERA is to help the user to analyze his problems through the visualization of their geometrical properties. It determines several different parallel solutions and their main characteristics are automatically computed as functions of the parameters : the processor count, the latency, and the number and the nature of the communications (local neighbor-to-neighbor or broadcast communications).

The paper is organized as follows. Section 2 shortly presents the method of PARE parallelization. An overview of OPERA is given in section 3. Section 4 illustrates its use with the Gaussian Elimination algorithm. Finally section 5, as a conclusion, compares OPERA with other existing tools and describes its current developments.

2 PARALLELIZATION OF PARAMETERIZED AFFINE RECURRENCE EQUATIONS

Among the many works concerned with DO-Loops parallelization, we distinguish two approaches. The first one is based on the direct parallelization of DO-Loop programs using various dependence tests (Zima and Chapman (1990), Pugh (1992), Banerjee (1993)). The second approach consists in first transforming the DO-Loops into single-assignment loop nests (Cytron et al. (1991)), which are free of dependences due to the lexicographical order of loop indices. These single assignment programs are then parallelized using synthesis techniques developped by the systolic community. We focus on this second step and deal with single assignment loops defined by systems of parameterized affine recurrence equations (PARE) defined over bounded convex polyhedra.

Example. Gaussian Elimination. In order to solve a linear system of equations $Ax = b$ where $A = (a_{ij})$ is an $n \times n$ matrix and b an n vector, the Gaussian Elimination triangulates the extended matrix $\mathbf{A} = [A \mid b]$. It is defined by the following loop nest :

```
DO k = 1 TO n-1
   DO j = k+1 TO n+1
      DO i = k+1 TO n
         A[i,j] = A[i,j] - A[i,k] * A[k,j] / A[k,k]
      DONE
   DONE
DONE
```

This loop nest can be transformed into the following equivalent single-assignment loop. The corresponding system of PARE given as input to OPERA is given figure 1.

*OPERA is a french acronym for "Toolbox for the Parallelization of Systems of Affine Recurrence Equations".

```
### Gaussian Elimination algorithm ###
#dimension of the problem
   DIMENSION 3
#parameter used in the visualization
   PARAM n=5
#input
   M[i,j] = DATA    { 1 <=i<= n, 1 <=j<= n+1 }
#output
   OUTPUT = A[i,j,k] { 1 <=i<= n , i <=j<= n+1 , k = i-1 }
#initialization equation
   A[i,j,k] = M[i,j] { 1 <=i<= n, 1 <=j<= n+1, k = 0 }
#computation equation
   A[i,j,k] = A[i,j,k-1] - A[i,k,k-1] * A[k,j,k-1] / A[k,k,k-1]
     { k+1 <=i<= n , k+1 <=j<= n+1 , 1 <=k<= n-1 }
```

Figure 1 Input file for the Gaussian Elimination algorithm.

```
DO k = 1 TO n-1
   DO j = k+1 TO n+1
      DO i = k+1 TO n
         A[i,j,k] = A[i,j,k-1] - A[i,k,k-1] * A[k,j,k-1] / A[k,k,k-1]
      DONE
   DONE
DONE
```

A parameterized affine recurrence equation has the following form:

$$X[z] = f\Big(\cdots, Y[g(z)], \cdots\Big) \qquad z \in D_p \tag{E}$$

where X and Y are array variables, f is any computation function with O(1) complexity, $p \in \mathbf{Z}^m$ is the vector defining the m parameters of the problem, $D_p \subset \mathbf{Z}^d$ is a convex bounded polyhedron associated with equation (E) called the *index domain* or the *iteration space*. It is defined by a set of linear constraints expressed by a system of linear inequalities $D_p = \{z \in \mathbf{Z}^d \mid P \cdot z \leq Q \cdot p + q\}$ where P is a $c \times d$ integer matrix, Q is a $c \times m$ integer matrix and q is a c integer vector. The *index mapping* g is an affine function with constant coefficients from $\mathbf{Z}^d$ to $\mathbf{Z}^{d'}$ ($d' \leq d$) of the form: $g(z) = R \cdot z + r$ where R is a $d' \times d$ integer matrix called the *index matrix* and r is an integer vector of size d'.

2.1 Dependence modeling and space×time mapping

In order to determine parallel solutions from such a PARE, the dependences have first to be modeled. This relies on the notion of *utilization set*. When considering a variable $Y[z_0]$, we call utilization set and we denote it by $Util_E(Y, z_0)$ the set of all the points of the domain using $Y(z_0)$ in their calculation. It is defined by :

$$Util_E(Y, z_0) = \{z \in D_p \mid g(z) = z_0\}.$$

Mongenet et al. (1991) show that this is a convex bounded polyhedron whose linear space is $Ker(R)$ where R is the index matrix. The dimension of the utilization set is therefore

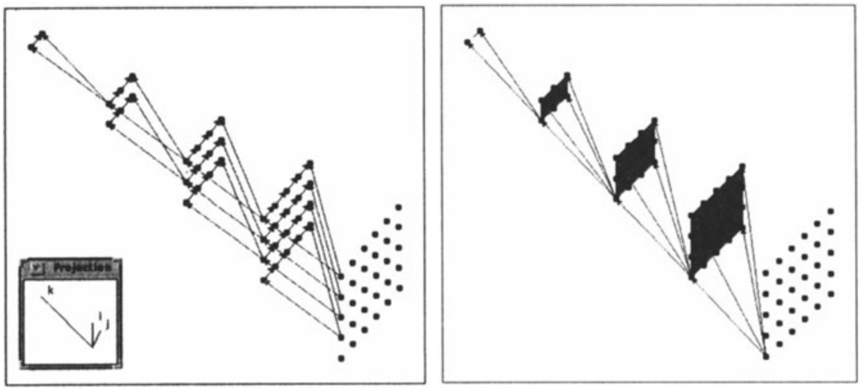


Figure 2 Domain and utilization sets $Util_2$ and $Util_4$.

$d - \boldsymbol{Rank}(R)$ where d is the dimension of the domain. The basis vectors of $Ker(R)$ characterize the utilization set and are called the *utilization vectors*. They are denoted by $u_{E,Y,i}$ ($i = 1, \ldots, d - \boldsymbol{Rank}(R)$). Notice that when R is a full row rank matrix, the utilization set is reduced to a single point and does not require any utilization vector.

The causal dependence associated with a variable Y is classically expressed by *dependence vectors* $d_{Y,g(z)} = z - g(z)$. Mongenet et al. (1991) show that the dependence vectors related to a variable $Y(z_0)$ define a *cone* characterized by its finite set of *extremal vectors*. These extremal vectors are denoted by $d_{Y,z_0,ex_i}, i \in \mathbf{N}$.

All points $z_0 = g(z)$ which are origin of a dependence vector d_{Y,z_0} form a set. This set is called the *emission set* and defined by :

$$\boldsymbol{Emit}_E(Y) = \{z_0 \in D_p \mid \exists z \in D_p \ such \ that \ g(z) = z_0\} = g(D).$$

Example. The Gaussian Elimination. In the computation equation there are four references to elements of variable A. We refer to these different references as respectively the first, second, third and fourth argument of the computation function and we index the corresponding index functions, index matrices, dependence vectors, utilization sets and utilization vectors accordingly.

The first argument corresponds to a uniform dependence characterized by index matrix $R_1 = Id$. The utilization set of any variable $A[z_0]$ has dimension 0 and the dependence is therefore characterized by a constant dependence vector $d_1 = (0, 0, 1)$.

The second and third arguments are both characterized by one-dimensional utilization sets since $\boldsymbol{Rank}(R_2) = \boldsymbol{Rank}(R_3) = 2$. The corresponding utilization and dependence vectors are respectively :

$u_2 = (0, 1, 0)$ $\quad d_2 = (0, j - j_0, 1)$ with $1 \leq j - j_0 \leq n - j_0 + 1$
$u_3 = (1, 0, 0)$ $\quad d_3 = (i - i_0, 0, 1)$ with $1 \leq i - i_0 \leq n - i_0$

The fourth argument induces two-dimensional utilization sets since $Rank(R_4) = 1$, and we have : $u_{4,1} = (1, 0, 0)$, $u_{4,2} = (0, 1, 0)$ and $d_4 = (i - i_0, j - j_0, 1)$ with $1 \leq i - i_0 \leq n - i_0$ and $1 \leq j - j_0 \leq n - j_0 + 1$.

The one-dimensional utilization sets $Util_2$ and the two-dimensional utilization sets $Util_4$

are presented in figure 2. The dotted vectors associated with each of these sets correspond to the extremal dependence vectors. The plain vectors correspond to the utilization vectors. □

The interest of considering such PAREs, i.e. single assignment programs, is the simple dependence modeling they induce. Hence, the dependence graphs have their vertices aligned on integer points of convex polytopes. The parallelization applies then on a dependence graph a space×time transformation. The transformations classically used are linear. They consist in particular in :

- An affine **schedule function** of the form $t : D_p \subset \mathbf{Z}^d \longrightarrow \mathbf{N}$ with $t(z) = \lambda \cdot z + \alpha$. It defines successive fronts on the domain which are the set of hyperlanes orthogonal to the *schedule vector* λ. The causal constraints are classically expressed by a system of constraints on the schedule function and the dependence vectors. For a system of PARE it is expressed by $\lambda \cdot d_{ex} > 0$ for all the extremal dependence vectors d_{ex} associated with the problem. Among the extremal vectors we call *minimal extremal vectors* the extremal vectors whose schedule component $\lambda \cdot d_{ex}$ is minimal. We denote them by d_{min}.

- A linear **allocation function** *alloc* : $\mathbf{Z}^d \longrightarrow \mathbf{Z}^{d-1}$ projects the domain along a unimodular vector denoted by $\xi \in \mathbf{Z}^d$ and called the *allocation* or *projection direction*. It results in a set of virtual processors that can be described by a DOALL loop program as given hereunder. All the points of D belonging to a given line directed by ξ define a segment called an *allocation segment* and are projected and executed in the same virtual processor, i.e. executed in the same iteration of the corresponding DOALL loop. Recall that the schedule and allocation functions must garanty that a virtual processor executes at most one computation at any given time step. This requires the condition $\lambda \cdot \xi \neq 0$ to hold.

Once a schedule and an allocation functions have been determined, the synthesis of a parallel solution consists in applying the corresponding linear mapping to the original specification of the problem. It transforms the problem expressed in the original domain D_p into an equivalent problem defined in a space×time domain. A virtual parallel solution can therefore be given in terms of loops characterized by one sequential loop defining the time and a set of DOALL loops characterizing the processors.

Example. For the Gaussian Elimination, a parallel solution can be synthesized with the schedule $\lambda = (1,1,1)$ and $\xi = (1,0,0)$ as allocation direction. We get the following virtual code usually refered as a SPMD code :

```
DOALL P(x,y) with 1 ≤ y ≤ n-1 , y+1 ≤ x ≤ n+1
  DO t = x+2y-3 TO x+y+n-4
     A[t,x,y] = A[t-1,x,y-1] - A[t-1,y,y-1]×A[x+2y-1,x,y-1]/A[3y-1,y,y-1]
  DONE
DONE
```

This loop can not be transformed into a non single-assignment loop by removing the inner index corresponding to the time variable. This is due to the use of `A[x+2y-1,x,y-1]` and `A[3y-1,y,y-1]` for the computation of `A[x,y]` at time `t` on processor `P(x,y)`. Since the source depends on the processor's reference and not on the time index `t`, the first index of `A` can not be removed in `A[x+2y-1,x,y-1]` and `A[3y-1,y,y-1]`. These references must be defined explicitly by communications and local storage of the data. □

Such a virtual code does not explicit the communications between the virtual processors. Using OPERA, in particular its dependence modeling, the communications can be synthesized in order to get a parallel code with explicit communication primitives, as seen figure 5 for the explicit code corresponding to the above single assignment program of the Gaussian Elimination. We show in the next section how these communication operations can be automatically determined from the projection of the utilization and minimal extremal vectors.

2.2 Communication synthesis

OPERA analyzes, for a given problem, how the choice of a particular linear space×time mapping influences the communications between the virtual processors of the parallel solution : the number of communications and their nature. We particularly focus on two types of communications : the local neighbor-to-neighbor communications and the broadcast communications. The results presented hereunder are developped by Mongenet (1995).

When dealing with communication synthesis, a user may focus on various constraints such as the nature or the number of the communication primitives. OPERA may help the user to deal with these two questions :

- **Broadcast utilization.** Depending on whether or not we are interested in parallel solutions with broadcast, we must add an appropriate constraint to the set of causal constraints in order to get a schedule. If we want the solution to be free of any broadcast, i.e. no variable can be used at a given time step by several processors of the DOALL loop, then we add the constraint $\lambda \cdot u_Y \neq 0$ for all the utilization vectors u_Y of the problem. Conversely if we want to take advantage of some broadcast primitives, then the constraint $\lambda \cdot u_Y = 0$ should hold for as many utilization vectors u_Y as possible.

- **Communication minimization.** In order to minimize the number of communications, one must try to localize as many arguments as possible. This can be achieved by choosing as allocation direction a vector projecting a utilization set on a minimal number of virtual processors of the DOALL loop. For a one-dimensional utilization set characterized by a unique utilization vector u the communication is optimally reduced with allocation direction $\xi = u$. Hence all the corresponding utilization points are projected on the same virtual *utilization processor* and this dependence requires at most one communication : from the virtual emission processor (the processor where is implemented the corresponding emission point) to the virtual utilization processor. If the emission point belongs to the linear space containing the utilization set, then no communication is required. For a two-dimensional utilization set characterized by two utilization vectors u_1 and u_2, one must choose an allocation direction parallel to the plane defined by u_1 and u_2 in order to project the plane on a single line of virtual processors.

3 AN OVERVIEW OF OPERA

3.1 Structure of OPERA

OPERA runs on *X/XView* environment. It is compiled with the *gcc* compiler, and *flex* and *yacc* for the parser. Polyhedra are represented using the Polyhedral Library developped at IRISA by Wilde (1993).

OPERA takes as input a system of parameterized affine recurrence equations defining the problem. The syntax of such a specification is similar to the ALPHA language proposed by Le Verge et al. (1991) as it was shown in figure 1 for the Gaussian Elimination. Recall that OPERA deals with parameterized equations, and therefore all the computations are realized using the parameters without instanciating them.

OPERA visualizes a synthesized solution by a grid of virtual processors (which can be described by a DOALL loop) and by the communication links between these processors. It is composed by the following main modules.

- the dependence modeling module determines the parameterized utilization sets as well as the extremal dependence vectors.
- the schedule determination module solves the system of linear inequalities $\lambda \cdot d > 0$. Depending on the user choice regarding broadcast utilization, the extra constraints $\lambda \cdot u \neq 0$ may be added or not.
- the allocation determination module. This module currently requires the user to choose an allocation direction. This will eventually be replaced by the automatic determination of several efficient allocation directions, such as the directions minimizing the number of communications (cf. Mongenet (1995)).

These different modules implement algebraic operations on convex parameterized polyhedra. In order to help the user to better apprehend the geometrical nature of his problem, each of these modules is coupled with a visualization module as shown in figure 3.

3.2 Algebraic tools on parameterized polyhedra

OPERA mainly manipulates convex parameterized polyhedra to represent the domains, the utilization sets, the fronts. It uses the two dual representations of a polyhedron :

- the *implicit representation* : the polyhedron is defined by a set of linear constraints.
- the *Minkovsky representation* also called the *parametric representation* by Schrijver (1986) : the polyhedron is characterized by a set of lines, rays and vertices.

To go from one representation to the other, we use the algorithm implemented in the Polyhedral Library. It is based on the Chernikova algorithm, successively improved by Fernández and Quinton in 1988 and Le Verge in 1992. Its complexity is $O(c^{\lfloor \frac{d}{2} \rfloor})$ where c is the number of constraints and d the dimension of the polyhedron.

The parameterized domains manipulated by OPERA are defined by the following implicit form :

$$D_p = \{z \in \mathbf{Z}^d \mid P \cdot z \leq Q \cdot p + q\}$$

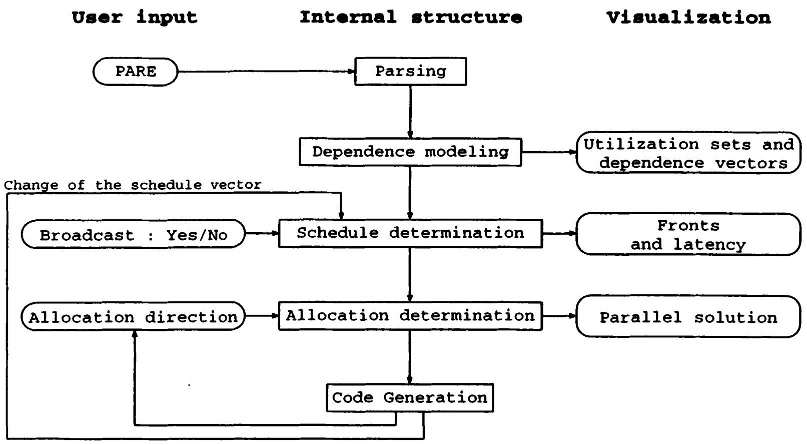


Figure 3 Overview of OPERA.

where P is a $k \times d$ integer matrix, Q is a $k \times m$ integer matrix, q is a k integer vector, or equivalently by the Minkovsky form :

$$D_p = \{z \in \mathbf{Z}^d \mid z = L \cdot \rho + R \cdot \mu + V_p \cdot \nu \mid \quad \mu, \nu \geq 0, \quad \sum \nu = 1\}$$

where L is the matrix whose columns define the lines of the polyhedron, while the columns of R are the rays and the columns of V_p are the vertices. ρ, μ, ν are free valued integer vectors.

In order to be able to use the Polyhedral Library and its revisited Chernikova algorithm, OPERA represents the parameterized polyhedra in the combined data and parameter space as follows :

$$D' = \{\begin{pmatrix} x \\ p \end{pmatrix} \in \mathbf{Z}^{d+m} \mid P' . \begin{pmatrix} x \\ p \end{pmatrix} \leq q\}$$

where $P' = [P \mid -Q]$.

Using the Minkovsky representation of D', we can find its faces of geometric dimension m called m-faces [†]. The vertices of the parameterized polyhedron D_p correspond to m-faces of its associated polyhedron D' in the combined space. To compute the parameterized vertex $v(p)$ corresponding to a given m-face, the algorithm described by Loechner and Wilde (1995) computes two affine transformations related to this m-face :

- the first one projects any point of the m-face in the data space $\mathbf{Z}^d$. Let $Proj_d$ be its associated matrix.
- the second one projects any point of the m-face in the parameter space $\mathbf{Z}^m$. Let $Proj_p$ be its associated square matrix. If matrix $Proj_p$ is singular then this m-face does not correspond to a vertex of D_p (proof given by Loechner and Wilde (1995)).

[†] More generally for any convex polyhedron, the faces of geometric dimension 0 define its vertices, the faces of dimension 1 define its edges, ...

Using these two transformations, each vertex $v(p)$ is defined by :

- a domain E corresponding to the projection of the m-face on the parameter space. It defines the restriction on the values of the parameters for which the corresponding vertex does exist.
- a $d \times m$ matrix $S = Proj_d \cdot Proj_p^{-1}$. The vertex $v(p)$ is only defined for $p \in E$ and its coordinates are $S \cdot p$. This information is used in OPERA to compute the extremal dependence vectors, as well as the latency, as function of the parameters.

4 THE GAUSSIAN ELIMINATION ALGORITHM

From a system of PARE, OPERA automatically computes the parameterized domains and visualizes them for a specified value of the parameters. It also performs the dependence modeling and exhibits it both algebraically and geometrically.

A schedule vector must satisfy the causal constraints $\lambda \cdot d_{ex} > 0$ for all the extremal vectors of the problem. Moreover to be free of broadcast, the constraint $\lambda \cdot u \neq 0$ should hold for all the utilization vectors of the problem. For the Gaussian Elimination $\lambda = (1,1,1)$ results in a parallel solution using only local communications. Moreover one can show that, in this case, it is the optimal affine schedule with a latency equal to $\tau = 3n - 4$.

If we allow solutions with one-dimensional broadcast communications, we may symmetrically broadcast the second or third argument, which correspond both to one-dimensional utilization sets. Either choice will result in a one-dimensional broadcast of the fourth argument because the corresponding utilization sets are two-dimensional and directed by $u_{4,1} = u_3$ and $u_{4,2} = u_2$. These two symmetrical solutions correspond to $\lambda = (0,1,1)$ and $\lambda = (1,0,1)$.

One may implement even more broadcast by choosing a schedule vector orthogonal to both u_2 and u_3, i.e. $\lambda = (0,0,1)$. This choice results not only in the broadcast of the second or third argument, but also in the full broadcast of the fourth one, due to the equality of the utilization vectors mentioned above. Notice that this schedule vector may be deduced from the observation of figure 2. Hence to get a 2-dimensional broadcast, one must schedule all the points of the grey planes in the rightmost window at the same instant and therefore choose a schedule vector orthogonal to these planes.

Let us now show for the schedule vector $\lambda = (1,1,1)$ how the parallel code with explicit communication primitives is synthesized. The number of communications can be reduced by localizing for example the third argument. In this case we have to choose $\xi = u_3 = (1,0,0)$ as allocation direction [‡]. Moreover since $u_{4,1} = (1,0,0)$ the fourth argument characterized by two-dimensional utilization sets is also localized on one dimension. The corresponding allocation function is $alloc(i,j,k) = (j,k)$.

The leftmost window in figure 4 shows how the utilization sets $Util_1$ and $Util_3$ are projected. The uniform dependence associated with $Util_1$ results in a local communication along the k-axis. Since the dependence related to $Util_3$ is localized, the only communication is generated by $d3_{min}$. The projection of $d3_{min}$ is similar to the one of d_1 : $alloc(d_1) = alloc(d3_{min}) = (0,1)$. Therefore these two dependences result in identical com-

[‡]Notice that the symmetrical solution $\xi = u_2 = (0,1,0)$ would give analogous results with the localization of the second argument instead of the third.

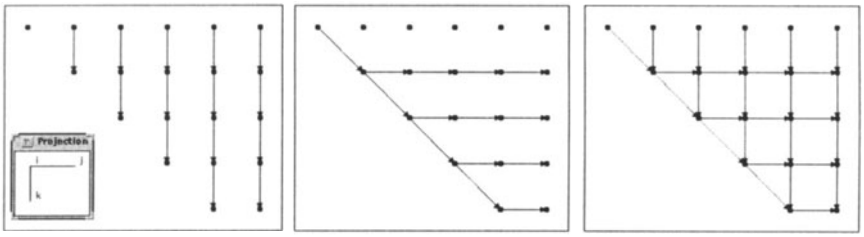


Figure 4 Communication synthesis with $\lambda = (1,1,1)$ and $\xi = (1,0,0)$.

munications from processor $P_{x,y}$ to processor $P_{x,y+1}$ [§]. Because the two corresponding emission sets are disjoint, these two dependences require only one communication per time step : the first time according to $Util_3$ to transmit the third argument and then according to $Util_1$ to successively transmit the different values of the first argument.

Utilization sets $Util_2$ and $Util_4$ require only two communications as shown in the middle window in figure 4. The diagonal arrows correspond to the projection of the minimal extremal vectors $d4_{min}$ and $d2_{min}$: $alloc(d4_{min}) = alloc(d2_{min}) = (1,1)$. The horizontal arrows correspond to the projection of the utilization vectors u_2 and $u_{4,2}$: $alloc(u_2) = alloc(u_{4,2}) = (1,0)$. Notice that the two corresponding emission sets are also disjoint, therefore they only require one communication per time step. Since utilization vector $u_{4,1}$ is parallel to ξ the fourth argument is localized on one dimension, besides being transmitted along the j-axis.

The corresponding parallel solution is visualized in the rightmost window of figure 4. The corresponding parallel code is given figure 5[¶] : in the communication operations the processor references are defined in the `Send` primitives. The processor references in the `Receive` primitives are only given as comments for the reader.

5 COMPARISON WITH OTHER WORKS AND FUTURE DEVELOPMENTS OF OPERA

OPERA's objective is not to capture the general program parallelization problem, like large systems tackling complex real-life applications such as SUIF (Amarasinghe et al. 1995). We deal with the problem of DO-Loop parallelization and focus on a formal version of single-assignment loops : the systems of affine recurrence equations. Since OPERA manipulates single assignment statements, it does not have to analyze memory access conflicts and therefore the dependence analysis is not based on one of the dependence abstractions proposed in the literature, such as the Dependence Distance or the Dependence Direction Vector. In systems of recurrence equations, the dependences are restricted to

[§]In the following the parallel solutions are visualized in a 2-dimensional space corresponding to the 2-dimensional DOALL loop given in section 2.1. We call x the horizontal axis and y the vertical one and we denote the processors by $P_{x,y}$.

[¶]The main objective of this description of a parallel solution is to focus on communications. It does not describe the input and output of data.

```
Processor $P(x,y)$
($x = y + 1$ and $1 \leq y \leq n - 1$)
    Receive $a_3$ <- $P(x,y-1)$
    Receive $a_4$ <- $P(x-1,y-1)$
    $R_3 = a_3$
    $R_4 = a_4$
    Send $a_4$ to $P(x+1,y)$
    Loop Receive $a_1$ <- $P(x,y-1)$
         Receive $a_2$ <- $P(x-1,y-1)$
         $a = a_1 - a_2 * R_3/R_4$
         Send $a_2$ to $P(x+1,y)$
         Send $a$ to $P(x+1,y+1)$
    EndLoop
```

```
Processor $P(x,y)$
($y + 1 < x \leq n + 1$ and $1 \leq y \leq n - 1$)
    Receive $a_3$ <- $P(x,y-1)$
    Receive $a_4$ <- $P(x-1,y)$
    $R_3 = a_3$
    $R_4 = a_4$
    Send $a_4$ to $P(x+1,y)$
    Loop Receive $a_1$ <- $P(x,y-1)$
         Receive $a_2$ <- $P(x-1,y)$
         $a = a_1 - a_2 * R_3/R_4$
         Send $a_2$ to $P(x+1,y)$
         Send $a$ to $P(x,y+1)$
    EndLoop
```

Figure 5 Explicit parallel code for $\lambda = (1,1,1)$ and $\xi = (1,0,0)$.

true dependences and the corresponding modeling is realized in OPERA using the notion of utilization sets.

Because OPERA manipulates single-assignment statements, it does not require symbolic analysis such as loop induction variable identification or loop invariant determination. Since it focuses on fine-grain parallelism, it is not concerned with interprocedural analysis or other techniques related to coarse-grain parallelism. Nevertheless, OPERA as a fine-grain parallelism analyzer could in the future be embedded in a larger system.

Its objective is to deal with problems with affine dependences (vs. uniform ones such as in Bouclettes, Boulet et al. 1994) and to automatically compute a set of parallel solutions for such problems. Among the different parallel solutions, OPERA determines the solutions satisfying some precise criteria such as optimal latency, minimization of the communications, minimization of the number of virtual processors. Communications minimization and locality optimization are determined according to various architectural constraints : we focus not only on local communications, but also on broadcast ones. Proper choices of affine schedules and allocations result in parallel solutions characterized by an efficient use of broadcast primitives if such primitives are available on the target architecture.

To help the user to choose among the many solutions to a problem, we believe that the vizualization tools offered by OPERA allow him to better understand his problem through a geometrical representation of its domain and its dependences. He may successively select various architectural criteria, compare the corresponding solutions in terms of processor count, latency or communication volume and choose the most convenient one.

Compared with other tools dealing with affine problems, such as COMPAR (Arzt et al. 1992), OPERA offers a *full parametrization* of a problem, its domains and dependences, and therefore results in parameterized solutions. These solutions are expressed by loops whose index bounds and array references are defined as functions of the parameters, as well as their latency. This is realized using Loechner and Wilde's representation of the domains and dependences by parameterized polyhedra, and the computation of their vertices as function of the parameters. Moreover, since OPERA uses theoretical results on polyhedra and linear algebra, it guaranties the correctness of the synthesized parallel solutions.

The current developments in OPERA are concerned with the automatic computation of the allocation directions in accordance to the schedule and to the characteristics of the target architecture. Our next step will be to implement the code generation. We aim at producing either data-parallel code or message passing code similar to the examples given in this paper. We are also working on alignment techniques in the context of affine dependences. This implies, when focusing on problems defined by several variables, the determination of affine-by-variable schedule and allocation functions.

REFERENCES

Amarasinghe, S.P., Anderson, J.M., Lam, M.S. and Tseng C.W. (1995) The SUIF Compiler for Scalable Parallel Machines. *Proceedings of the seventh SIAM Conference on Parallel Processing for Scientific Computing.*

Arzt, U., Teich, J. and Thiele, L. (1992) The Concepts of COMPAR - A Compiler for Massively Parallel Architectures. *Proceedings of IEEE ISCAS, San Diego*, 681–4.

Banerjee, U. (1993) Loop Transformations for Restructuring Compilers, the Foundations. *Kluwer Academic Publishers.*

Boulet, P., Dion, M., Lequiniou, E. and Risset, T. (1994) Reference Manual of the Bouclettes Parallelizer. *Technical report 94-04, ENS-Lyon, France.*

Cytron, R., Ferrante, J., Rosen, B., Wegman, M. and Zadeck, K. (1991) An Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, **13:4**, 451–90.

Fernández, F. and Quinton, P. (1988) Extension of Chernikova's Algorithm for Solving General Mixed Linear Programming Problems. *Technical Report 437, IRISA, Rennes.*

Le Verge, H., Mauras, Ch. and Quinton, P. (1991) The Alpha Language and its use for the Design of Systolic Arrays. *Journal of VLSI and signal processing*, **3**, 173–82.

Le Verge, H. (1992) A note on Chernikova's Algorithm. *Technical Report 635, IRISA, Rennes, France.*

Loechner, V. and Wilde, D. (1995) Parameterized Polyhedra and their Vertices. *Technical Report 95-16, ICPS, Université Louis Pasteur, Strasbourg, France.*

Mongenet, C., Clauss, Ph. and Perrin, G.R. (1991) A Geometrical Coding to Compile Affine Recurrence Equations on Regular Arrays. *5th International Parallel Processing Symposium, IPPS'91, Anaheim, California.*

Mongenet, C. (1995) Mappings for Communication Minimization using Distribution and Alignment. *Conf. on Parallel Architectures and Compilation Techniques, Limassol, Cyprus*, 185–93.

Pugh, W. (1992) A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM, Aug. 1992*, 102–14.

Schrijver, A. (1986) Theory of Linear and Integer Programming. *John Wiley and Sons, New-York.*

Wilde, D. (1993) A Library for doing Polyhedral Operations. *Technical Report 785, IRISA, Rennes, France.*

Zima, H.P. and Chapman, B. (1990) Supercompilers for Parallel and Vector Computers. *ACM Press, Addison Wesley.*

13
Program Comprehension Engines for Automatic Parallelization: A Comparative Study

Beniamino Di Martino[*] Christoph W. Keßler[†]

Abstract

We compare two systems for program comprehension that are targeted towards support of automatic parallelization: the PAP recognizer currently included into the Vienna Fortran Compilation System, and the PARAMAT pattern recognizer developed at Saarbrücken University. We illuminate the main differences, the advantages and disadvantages of each approach, and show how both approaches may be integrated to combine the generality of one approach with the speed of the other one.

1 Introduction

Program comprehension is the process of discovering abstract concepts in the source code. In its generality, it does not seem automatable; but, given some knowledge on the program's application domain, automatic understanding is possible at least on a local level. In this case, understanding becomes a recognition process: it can be sketched as matching a given source program against a set of known programming patterns.

A number of problems have to be dealt with when facing automated recognition of algorithmic patterns [16]. The most important ones are *syntactic variation*, *algorithmic variation* (a pattern can be implemented in many different ways), *delocalization* (the implementation of a pattern may be spread throughout the code), and *overlapping implementations* (portions of a program may be part of more than one pattern instance).

Automatic program comprehension systems are mainly used for two purposes: to support software maintenance, e.g. for automatic documentation of code, and to support automatically parallelizing compilers. Several methods for both areas have been proposed within the last years, and some (mostly experimental) systems have been built. Here we focus on the second issue.

Automated program recognition can play a crucial role in overcoming limitations of existing tools for automatic parallelization for distributed-memory architectures. For instance, replacement of recognized sequential code by suitable parallel algorithms overcomes an important limitation of existing automatic parallelizers that are able to parallelize some loops but are still bound to the sequential program's control structure. Moreover, the acquired knowledge enables automatic selection of sequences of optimizing transformations, supports automatic array alignment and distribution, and improves accuracy and speed of performance prediction [8].

[*]Inst. for Software Technology and Parallel Systems, University of Vienna, Austria, and DIS - University of Naples, Italy. email: `dimartin@par.univie.ac.at`

[†]FB 4 Informatik, University of Trier, D-54286 Trier, Germany. email: `kessler@psi.uni-trier.de`

The application domain considered mainly consists of numerical computations, in particular linear algebra and PDE codes. Domain analysis [8] has shown that the size of the set of patterns typically occurring in such codes remains reasonably small. But also in non-numerical or irregular numerical fields, recognition of algorithmic patterns in the code can drive the selection of suitable parallelization strategies, as [4, 5] shows for the *Divide-and-Conquer* pattern in Quicksort implementations and the *Branch-and-Bound* pattern in optimization codes.

We present two automatic program comprehension engines that have mainly the same goals but vary considerably in their methods, properties, and implementations. We compare these approaches in detail, showing their advantages and disadvantages, relate them to earlier work and conclude with a proposition to combine the speed of one method with the flexibility of the other one.

2 Deterministic Program Recognition in PARAMAT

PARAMAT's pattern recognizer works on the intermediate representation of the source program as an abstract syntax tree. A well-structured and statically analyzable source language is assumed. The goal is to annotate as many nodes as possible with a so-called *pattern instance*, a summary structure that describes which function is computed in the subtree rooted at that node, together with the parameter objects of that function. Speed and robustness of this method mainly result from exploiting the natural semantic hierarchy of the patterns in the library.

Preprocessing First we apply several normalizing transformations to make the program as explicit as possible, by inlining all procedures (recursive procedures are very untypical for the application area considered), forward propagation of constant expressions, recognition and replacement of induction variables, and eliminating dead code.

Semantic hierarchy of patterns Each pattern consists of a specification of a (mathematical) operation and of the types and the data structures of its parameters.

For instance, the MV pattern represents the operation $\vec{y} = \mathbf{A}\vec{b}+\vec{x}$, with the parameters $\vec{y}$, $\mathbf{A}$, $\vec{b}$, and $\vec{x}$ being real (sub-)arrays ($\vec{x}$ may also be a constant).

For each nontrivial pattern, we usually know by experience many implementation prototypes (for sequential C code). Because of the wide variety of semantics preserving code transformations, the number of such prototypes can be tremendous for more complex patterns (such as matrix–matrix multiplication), blowing up the size of an automatically generated tree automaton dramatically. For this reason, we formulate the prototypes as far as possible by using instances of (other) patterns. E.g., an implementation of matrix–vector product (MV) can be written as a single loop around a dot product

```
for (i=1; i<=n; i++)
   SSP( x[i], A[i][1:m], b[1:m], x[i]);
```

or as a loop summing up the result vectors of vector triads

```
for (j=1; j<=m; j++)
   VAADDSV( x[1:n], b[j], A[1:n][j], x[1:n]);
```

because $(\mathbf{A}\vec{b}+\vec{x})_{i=[1:n]} = (\sum_{j=1}^{m} A_{ij}b_j + x_i)_{i=[1:n]} = \sum_{j=1}^{m}((A_{ij}b_j)_{i=[1:n]})_j + (x_i)_{i=[1:n]}$.

With such domain information it becomes straightforward to formulate *templates*, that are the rules to determine a node's pattern m (and pattern instance I) given the node's operator and all its children's pattern (i.e., subpattern) instances. In the case of several children, we select for each template the most characteristic child, and call the expected pattern for this child the *trigger pattern*. For instance, there is a template for MV with trigger pattern SSP, and another one with trigger pattern VAADDSV. For each pattern, we realize only the most important templates matching it (typically, we have 1 to 3 realized templates per pattern), see [8] for the complete list.

This natural semantic pattern hierarchy is stored in a directed graph (*pattern hierarchy graph, PHG*) where the patterns are the nodes and for each template an edge goes from the trigger pattern to the pattern to be inferred. Thus pattern recognition becomes a path finding problem in the PHG. Besides from cycles from a pattern to itself, the PHG is acyclic. Different paths towards a pattern m correspond to different implementations of the functionality of m. Thus, a linear-sized PHG (and thus, pattern recognizer) represents exponentially many implementation variations of the same pattern.

The PHG has a second important advantage: it serves also as a hash table that can be inspected by the pattern recognition algorithm, as it yields immediately all possible superpatterns that could be matched from a given trigger pattern. Often, the trigger pattern together with the operator of the node to be matched suffices to select a single possible template. If there are several templates admissible, these are tested concurrently; the result is deterministic. Failing templates abort as soon as possible.

Recognition algorithm The abstract syntax tree is traversed from left to right in postorder. For a leaf node (a variable or a constant), it is trivial to determine its pattern (VAR or CONST, respectively). At each inner node v of the syntax tree, the algorithm tests, based on v's operator and v's children's patterns already matched, whether there is a pattern m in the library (there exists at most one) which matches the semantics of the subtree T_v rooted at v. Selection of admissible templates is done by inspecting the PHG. For each of them a short routine is called that realizes that template; it fails if it cannot prove that the function computed by T_v equals the operation represented by m, and returns an instance I of pattern m otherwise. In the latter case, it also maps the program objects to the slots of I (pattern parameters), and annotates v with I.

Before trying to match a new loop header, the algorithm distributes [17] that loop as far as possible and applies pattern matching to each of the resulting loop headers separately. Loop distribution is often supported by scalar expansion [17], using temporary arrays.

The basic method is extended for pattern matching along 'horizontal' dataflow edges, such that several (matched) instructions in the same block that belong to the same pattern may be contracted to a single pattern instance, even if they are textually separated. Several instructions may belong to the same thread of computation only if their operands are involved in at least one of several types of dataflow relations that we denote by dataflow edges. These *cross edges* in the syntax tree represent particular, loop-independent data flow relations among the operands of pattern instances within the same block. Thus they are well-suited to guide 'horizontal' pattern recognition [9]. Computing exact array data flow is generally hard, but here we can profit from the simple array access structures that are characteristic for dense matrix computations and that are present in all our patterns. Matching along cross edges is particularly helpful to disentangle intermixed computations, thus guiding pattern recognition, or to reroll unrolled loops.

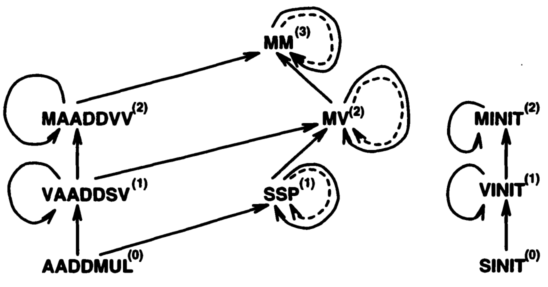
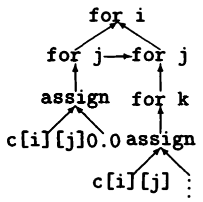


Figure 1: The pattern hierarchy graph of Matrix–matrix multiplication. Solid edges mean realized templates for 'vertical' pattern recognition; dashed ones for 'horizontal' pattern recognition along cross edges. Solid cycles mean templates for unblocking or elimination of semantically invariant conditionals; dashed cycles represent templates for loop rerolling or integration of initializers.

Example We demonstrate the pattern recognition algorithm at a simple example. Matrix–matrix multiplication is well suited since the functionality of it and of its subpatterns is widely known, and since its PHG (Figure 1) is rather clear. Suppose the following program fragment is given:

```
    for (i=1; i<=n; i++) {
      for (j=1; j<=m; j++)
S1:     c[i][j] = 0.0;
      for (j=1; j<=m; j++)
        for (k=1; k<r; k++)
S2:       c[i][j] = c[i][j]+a[i][k]*b[k][j];
    }
```



The algorithm traverses the abstract syntax tree from left to right in postorder. First, it encounters at S1 a scalar initialization SINIT (c[i][j], 0.0). For the j loop around it, we obtain an instance of a vector initialization VINIT(c[i][1:m], 0.0). The access to array c has become a vector as one dimension has been bound by the loop.

Next, assignment S2 is considered and annotated by AADDMUL (c[i][j], a[i][k], b[k][j], c[i][j]) (accumulative addition of a product). Following the suitable PHG edge, this yields a dot product for the k loop: SSP (c[i][j], a[i][1:r], b[1:r][j], c[i][j]). The accesses to the arrays a and b have become vectors. As the accumulating scalar c[i][j] has not been initialized so far, it has to be entered into the initialization slot of the SSP instance to keep data access information consistent. Then, the do j loop around the SSP instance is recognized as an instance of matrix–vector multiplication. Again the accumulating vector c[i][1:m] fills the initialization slot. The partially matched, unparsed syntax tree now looks as follows (code parts 'below' recognized nodes are not shown any more):

```
for (i=1; i<=n; i++) {
  VINIT(c[i][1:m], 0.0);
  MV(c[i][1:m],a[i][1:r],b[1:r][1:m],c[i][1:m]);
}
```

At this point we need data flow information in order to continue recognition. Here we obtain that vector c[i][1:m] is written in the VINIT instance, and read and overwritten by the MV instance. A *cross edge* of type FLOW symbolizes that data flows between these two instances in an expected way. This situation can be tested by a realization of another template for pattern recognition along cross edges. As the template matches, these two instances are merged into a single MV instance `MV (b[1:r][1:m], a[i][1:r], 0.0)`, i.e., the initialization slot is now filled by `0.0` from the VINIT instance. This instance, in turn, can be matched with the `i` loop into `MM(c[1:n][1:m], a[1:n][1:r], b[1:r][1:m], 0.0)` (matrix–matrix multiplication) representing this entire piece of code.

During pattern recognition, we have followed the PHG paths SINIT...VINIT and AADDMUL...SSP...MV...MV...MM. Common program transformations, like loop interchange or loop distribution, would result in a different path being taken towards MM, but would not prohibit pattern recognition.

Postprocessing Finally we must eliminate useless code that may emanate from conservative cross matching and certain transformations. Instances of *unstable patterns* are decomposed into their basic patterns' instances. E.g., general vector operations are decomposed into simple vector operations using temporary arrays.

Implementation The current prototype implementation consists of 12000 lines of C code and reliably recognizes 91 nontrivial patterns with 150 nontrivial templates. Each template realization is implemented as a C routine of 20 to 50 lines. Since many useful syntactic and semantic predicates have been predefined, writing code for templates is handy and straightforward. More patterns can easily be added. Robustness against loop interchange, loop distribution, loop unrolling and statement reordering has been exemplified in practice as well as the high speed of the recognition algorithm [8].

3 Backtracked program recognition: the PAP Recognizer

PAP (*Parallelizable Algorithmic Patterns*) Recognizer is a (prototype) tool for automated program comprehension, aimed towards support of code parallelization. It implements a *plan based* technique for the recognition of *concept instances* in the code, that works in a hierarchical way. It provides as output a graphical browser that visualizes the hierarchy of recognized concepts and their position in the source code. It is mainly driven by the semantic features (control, data dependence, calling relationships) of the concepts. This, combined with the backtracking feature of the recognition procedure, offers high flexibility that allows to handle non-numerical and irregular codes as well as numerical ones.

The prototype relies on Prolog as system shell, thus taking advantage of its deductive inference engine. It utilizes the structural analysis of the input code performed by the *Vienna Fortran Compilation System* [18] (VFCS) front-end, an interactive compilation system for distributed memory parallel computers, in which PAP Recognizer has been integrated as parallelization support tool [6].

Recognition process PAP Recognizer performs a hierarchical parsing process, driven by *concept recognition rules*, that acts on *concept instances* descriptions.

The matching rules of the hierarchical concept parsing (*plans*) are production rules that describe the features of the concepts to be recognized. Features identifying an (algorithmic) concept can be informally defined as the way some abstract functions (the composing subconcepts) are related and organized into a specific abstract control structure. By "abstract control structure" we mean structural relationships, such as control flow, data flow and calling relationships. These relationships involve "abstract" objects, i.e. aggregates of variables or statements linked by a functionality.

More specifically, each concept is recursively specified by its compositional hierarchy, and by relationships and constraints among composing subconcepts.

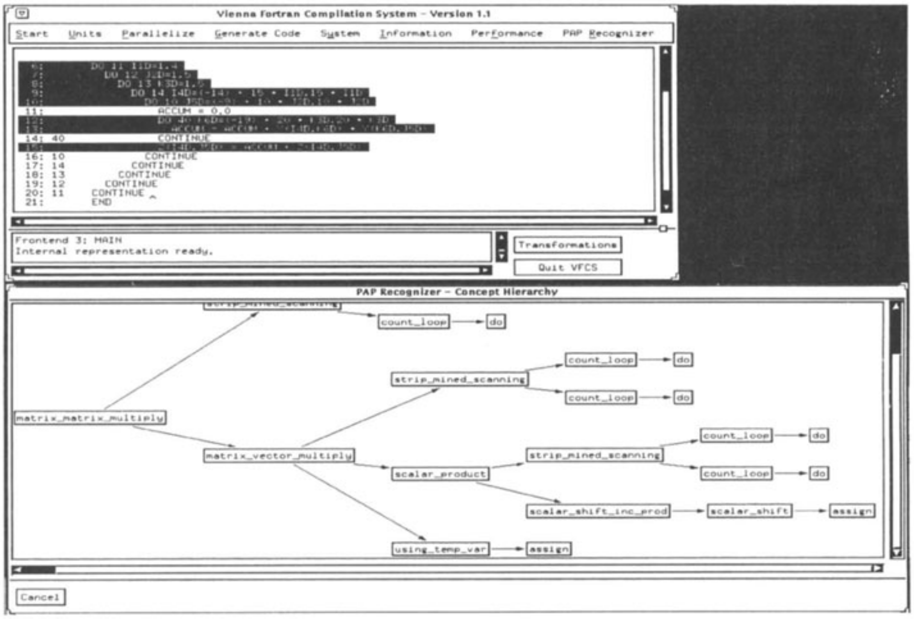
With regard to the control and data dependence relationships, to put in evidence their peculiarity and to simplify the concepts specification, they are subjected to an abstraction process during the recognition, likewise the concept abstraction. Indeed, a notion of *abstract* control and data dependency between abstract concepts has been introduced, and defined in a recursive way: a concept instance C_i is defined as *abstract data (control) dependent* on another concept instance C_j, if the composing subconcepts of C_i satisfy a particular pattern of abstract data (control) dependence relationships with the concept C_j; this pattern is characteristic of each concept C_i, and is determined by the plan which recognizes it.

The terminals of the recursive specification of abstract concepts, and abstract data and control dependence relationships among them, are represented by the set of syntactical, control and data dependences on the program, obtained by a structural analysis of it, and codified as *base* concepts and dependence relationships.

A top-down direction (demand driven recognition) has been chosen for the hierarchical parsing. This is due to the particular aim of the recognition process, that is trying to find instances of *Parallelizable Algorithimc Patterns* in the code (algorithmic patterns for which a (set of) parallelization strategies can be defined, related to the underlying architecture). For this purpose, a recognition of the functionalities of the whole program is not needed, and thus a bottom-up parsing is not necessary. If the list of high level PAPs to be searched in the code is well defined (especially in relation to the characteristics of the underlying architecture), and if the recognition plans are cleverly designed in such a way to fail as soon as possible, the top-down approach allows for a deep pruning of the search space associated with the hierarchical parsing.

Abstract program representation The output of the structural analysis phase, that is syntactic, control flow, data flow and data dependence informations on the program, is utilized to build the *Base Internal Representation* of the program, which is stored in the *Concept Instances Database* in the form of *base concepts*. This base level of the representation is substantially a Program Dependence Graph (PDG), whose nodes represent statements and whose edges represent control and data dependences. This slightly differs from a standard PDG graph, in that it is augmented with the following structures:

- control dependence edges are labeled with the branch (true, false) of the dependence, and data dependence edges are labeled with the dependence variable identifier and the kind of the dependence (loop independent or loop carried);
- assignment statement nodes are augmented with two tree structures representing its left and right hand sides; subscript expressions of array variable instances are likewise represented by tree structures, linked to the node representing the array variable instance;
- each control statement node is augmented with the tree structure representing the control conditional expression;
- each node of the augmented PDG points to the corresponding nodes of the syntax tree, so that a direct reference from the abstract concept recognized to the code implementing it is possible; in this respect, the PDG could be viewed as a kind of web;
- variable definitions are explicitly represented, including information about the type, the rank for array variables, and including pointers to the corresponding nodes of the



```
          DO 11 I1D=1,4
            DO 12 J2D=1,5
              DO 13 K3D=1,5
                DO 14 I4D=(-14) + 15 * I1D,15 * I1D
                  DO 10 J5D=(-9) + 10 * J2D,10 * J2D
                    ACCUM = 0.0
                    DO 40 K6D=(-19) + 20 * K3D,20 * K3D
                      ACCUM = ACCUM + X(I4D,K6D) * Y(K6D,J5D)
      40            CONTINUE
                    Z(I4D,J5D) = ACCUM + Z(I4D,J5D)
      10          CONTINUE
      14        CONTINUE
      13      CONTINUE
      12    CONTINUE
      11  CONTINUE
          END
```

```
Frontend 3: MAIN
Internal representation ready.
```

Figure 2: **Recognition of a tiled implementation of matrix–matrix multiplication**

syntax tree.

The overall internal program representation is generated during the recognition phase performed by the PAP Recognizer. It has the structure of a *Hierarchical PDG* (HPDG), reflecting the hierarchical strategy of the recognition process. Starting from the base internal representation, the recognizer performs the concept parsing, following the rules specified by the plans. As long as the parsing process proceeds and more and more abstract concepts are recognized, they are represented as nodes in increasingly higher layers of the HDPG. More specifically, each node representing a concept is linked to the nodes of the lower layer representing subconcepts. Abstract control and data dependence edges for the newly created abstract concept nodes are inherited from those of the composing subconcept nodes, in a way that is characteristic for each concept, and which is determined by the plan which recognizes it. Internal Program Representation is stored in the Concept Instances Database which is incrementally updated by the PAP recognizer as the recognition advances.

Features The syntactic variation problem is solved by: (1) characterizing inter–statement level concepts with non-syntactic properties like control and data dependence that are a better characterization of concepts than other lower level features; (2) taking advantage of the backtracking characteristic of the recognition procedure to perform symbolic analysis of expressions within statements. The delocalization problem is solved by the characteristics of the abstract program representation which: (1) is based on an inherently delocalized structural representation (PDG); (2) has a global scope of visibility, so that the active rule can attempt to match all instances of concepts already recognized, at every level of the abstraction. The implementation variation problem is solved by the backtracking feature of the recognition process. More specifically, backtracking allows the specification of one concept by means of multiple plans: each plan specifies a different algorithmic implementation of the same concept. Finally, the overlapping implementation problem is solved by the global scope of visibility of the representation, and by the fact that the parsing mechanism does not restrict the use of a subconcept to one plan, allowing the recognition even in presence of shared concept instances.

An important consequence of the features just discussed is the independence from restructuring techniques, that modify the original code before and during the recognition process to deal with delocalized code and implementation variations. This means that this approach does not need a canonical form for concept implementations.

Example We show how an instance of the matrix–matrix multiplication algorithmic pattern, implemented with tiling of the iterations, is recognized in a code fragment. Fig. 2 (upper window) shows the Fortran program segment normalized by the VFCS front end. This version is analyzed by the PAP Recognizer, and Fig. 2 (lower window) shows the hierarchical structure of the recognized pattern, in the form of a graph. By clicking on each of the graph nodes, representing a composing subconcept, it is possible to highlight the corresponding code implementing it, as shown in the figure for the overall concept instance. As can be seen in this figure, the tool is able to recognize the concept in the presence of a temporary accumulator variable for the scalar-product subconcept; the implementation variation represented by the tiling is recognized by composing the matrix multiplication concept with the concepts of strip mining and loop interchange, as can be seen in the figure representing the hierarchical structure.

4 Detailed Comparison

Determinism Both approaches are hierarchical. But there is a decisive difference:

The PARAMAT recognizer is deterministic: each syntax tree node can carry at most one pattern instance which summarizes anything that is contained in the subtree below it. This is made possible by applying loop distribution. Moreover, the PARAMAT recognition is leveled: Because at each level there is only a very limited number of candidates that may match the node, the speed of recognition is very high.

The PAP Recognizer applies backtracking: this allows for dealing with implementation variations and delocalization of concept instances. Moreover, the concept matching is not limited in the level of abstraction: the particular structure of the abstract program representation (the hierarchical abstract PDG) make it possible to have a global scope of visibility of the recognized subconcepts, thus allowing the recognition even in presence of sharing of concept instances. These two features make recognition more powerful, but also make the search complexity grow exponentially with the code size. Nevertheless, both the top down approach, that permits to not inspect all the code, and the summarization of derived subconcept within HPDG nodes, prune the search space considerably.

Program representation and representation of recognized features The PARAMAT recognizer uses the abstract syntax tree of the source program as intermediate representation. Data flow information is computed incrementally during the recognition

system	PAP Recognizer	PARAMAT pattern recognizer
direction	top–down	bottom–up
driving features	syntax and semantic driven (control, data dependence)	syntax driven (data flow used to deal with delocalization)
backtracking	yes, essential for the method	not necessary (deterministic)
restructuring	not necessary for the method	essential (canonicalization)
preprocessing	not necessary	necessary
power of recognition	more powerful (sharing, delocalization, variation of implementations)	less powerful
speed	slow	fast
goals	alignment and distributions selection; replacement by library routines only if coupled with restructuring; template based par. code generation	alignment and distributions sel., replacement by library routines

Table 1: Main differences between the two approaches

process and only where needed. If required, the syntax tree can be transformed during the recognition process. — A recognized code portion, here always identical to a subtree of the abstract syntax tree, is summarized in a pattern instance which annotates the root of the matched subtree. The pattern instance contains all the information of what is computed in that subtree (but not any more *how* it is computed). A pattern instance I annotating a node v can be used for two purposes: (1) It can be used to determine the pattern of the father of v. There is no need of I for the recognition process any more after v's father has been also matched. (2) It offers the possibility of code replacement.

The PAP Recognizer starts using the PDG as intermediate representation. During the recognition process, it builds upon the PDG an abstract hierarchical PDG that represents the hierarchy of recognized concepts. Although associated with specific program locations and program objects, this abstract representation is not limited in its scope of visibility but globally visible, thus allowing for subconcept sharing and recognition of delocalized concepts. Each derived subconcept is summarized in its functionality, compositional hierarchy and linking to the syntax tree nodes (and thus to the code segments implementing it) by its node in the HPDG.

Recognition process The PARAMAT recognizer is driven by the syntax tree structure and by the data flow edges computed meanwhile. Each instance of a recognized pattern is one-to-one linked to a node in the abstract syntax tree. This makes pattern recognition deterministic and relates recognition to a traversal of the abstract syntax tree.

The PAP recognizer is driven by control and data dependence relationships of the (inter-statement level) concepts. The parsing process is not related to a syntax tree traversal; it is rule based and with backtracking feature. Rules are applied on a globally visible abstract program representation. Although this characteristic in principle increases the complexity of the process, the systematic use of control and data dependence relationships to characterize concepts allows the application of rules to be driven by the locality typically present in the source program. In this way complexity can be maintained at an acceptable level, without constraining the delocalized recognition capability.

Restructuring during recognition PARAMAT's recognizer relies on powerful restructuring transformations such as loop distribution or scalar expansion. This simplifies loop bodies and allows to apply a simpler pattern matching algorithm. The modifications of the source program are motivated by the application domain (replacement by parallel routines) and are compatible with the code replacement algorithm [10]. The most important transformation in this context is loop distribution because it allows to factor out different computations from the same loop nest that can be matched separately by different patterns (and thus result in separate code being generated).

The PAP Recognizer doesn't need to restructure during recognition because it is not based on canonicalization of concept implementations. Syntactic variations (as textual nesting) are instead as much as possible made transparent to the recognition process by using non-syntactic features like control and data dependence. For the same reason, the approach doesn't need (if not to speed up the recognition) preprocessing transformations, and can thus deal with situations where these cannot be applied.

Sharing The PARAMAT recognizer restructures the program and applies loop distribution to factor out the shared code portions as far as possible. It then decides (if not directed by data dependences, by using a heuristic) the order of pattern instances and assigns shared code portions to the first of them. The intermediate results of the shared code portions used by subsequent pattern instances must be written to temporary variables. This is necessary because the goal is code replacement. Simple duplication of common code is not provided; indeed that would lead to incorrect code in some cases.

The PAP recognizer has no problems with code portions shared by several patterns, because of the nonlevelled and nonlocalized matching mechanism, and because of the possibility to reuse derived subconcepts for other matchings.

5 Related Work

Several automatic program comprehension techniques have been developed over the last years. They vary considerably in their application domain, method, and status of implementation.

Earlier work targeted towards automatic code optimization or parallelization Snyder [15] addresses idiom recognition in APL codes. His algorithm is an extended depth-first traversal of the abstract syntax tree with linear expected run time. He applies dynamic programming techniques to select the most profitable idiom in the presence of overlapping idioms, which appears to be common in APL programs. — [3] suggests (non-constructively) to apply pattern matching techniques for the detection of reductions and recurrences within the framework of a formal system for automatic shared memory parallelization. — EAVE [2] is an expert system for interactive vectorization of FORTRAN programs. It contains a simple pattern matching tool that can decover order 1 patterns (vector operations, reductions). — The pattern matcher of [12] works on a modified program dependence graph that has been extended in a special way to match certain loop structures with the goal of replacing them by parallel algorithms. The cost of recognition is higher because the rewrite rules form a graph grammar. Normalization of the PDG has to be provided interactively by the user. — [13] proposes a special approach for recurrence detection. While this method offers, at considerable computational effort, the recognition of rather general and multidimensional recurrences, a number of assumptions are made that are hardly met by real applications. As complicated recurrences are rare in real programs, the computational effort of this approach seems not justified. — CMAX [14] is the only commercial application of pattern matching with regard to parallelization. It translates FORTRAN77 programs to CM-Fortran, a parallel vendor–specific Fortran dialect similar to Fortran90. It recognizes syntactically several common loop constructs (vector operations, reductions, matrix–matrix multiply) but does not distinguish between patterns and templates. The recognition power is slightly weaker than PARAMAT's, but the main advantage of CMAX is its ability to recognize FORTRAN-specific storage conventions and to transform them to make the program machine-independent and more suitable to distribution of data in that point. — Similar to the systems compared in this paper is the first phase of a program comprehension system for FORTRAN programs sketched in [11]. It works top–down on the PDG and partly uses the algorithm from [15].

Other problem domains Some systems for program comprehension in a non–numerical domain are targeted towards automatic documentation and support of software maintenance. Plan Calculus [16] represents code and patterns (called "clichés") with graph structures whose nodes correspond to subconcept instances and whose arcs capture control and data flow relationships among them. Clichés recognition becomes thus a *graph parsing* process using a set of *graph grammar rules.* It produces a parse tree representing a hierarchical description of plausible concepts of the program. — PAT [7] uses an abstract, object-oriented representation for syntactic and semantic concepts composing the program. Each concept is an instance of a concept class, and the classes are hierarchically structured. Our templates are roughly comparable to their "plans"; a plan's representation consists of a description of the syntactical components and a description of the constraints to be satisfied by components. An inferential pattern–directed engine derives new higher–level concepts from the existing ones, utilizing plans as inference rules.

6 Conclusion

We have presented two systems for automatic program comprehension, the PARAMAT pattern recognition tool and the PAP recognizer. Both systems are targeted towards comprehension of numerical codes (even though the flexibility of the second approach makes it suitable to deal with irregular or non-numerical code too) with the goal to support automatic parallelization.

Nevertheless, the systems vary considerably in their methods, properties, and implementations. While, roughly speaking, PARAMAT's pattern recognizer provides acceptable (for the given domain) recognition power at impressive speed and its output is suited for code replacement more straightforwardly, the PAP recognizer offers more flexibility and generality at the expense of higher run time. Thus, there is just a tradeoff between power of recognition and speed to be deliberated before choosing the one or the other system for a certain application program, depending on its size and complexity.

We are currently studying the possibility and effectiveness to combine both approaches. One possibility could be the following. If recognition time is very critical, we recommend to use PARAMAT's pattern recognizer for longer codes. For shorter codes, or if recognition time is less critical, the greater flexibility of the PAP recognizer should be exploited. If pure PAP recognition takes too much time and one is willing to trade recognition power for time, we propose that the PARAMAT pattern recognizer could be run as a preprocessor; the pattern instances produced become then facts for the PAP recognizer. Restructuring done by the PARAMAT recognizer is finished before the PDG for PAP recognition is constructed. The already recognized program parts are not submitted to PAP recognition any more, resulting in a downsized PDG. This corresponds to explicit a priori pruning of the PAP search tree and clearly limits the flexibility of the PAP recognizer. The degree of preprocessing could be modified by the user, e.g. by limiting PARAMAT's recognition to some restricted set of patterns. It is up to the user to decide about that when recognition time is critical.

References

[1] S. Bhansali, J.R. Hagemeister, C.S. Raghavendra, H. Sivaraman, "Parallelizing sequential programs by Algorithm-level Transformations", 3rd *IEEE Workshop on Program Comprehension*, Washington, Nov. 1994.

[2] P. Bose. Interactive Program Improvement via EAVE: An Expert Adviser for Vectorization. *Int. Conf. on Supercomputing*, pp 119–130, July 1988.

[3] T. Brandes, M. Sommer. A Knowledge-Based Parallelization Tool in a Programming Environment. *Int. Conf. on Parallel Processing*, pp 446–448, 1987.

[4] B. Di Martino, G. Iannello. Towards Automatic Parallelization through Program Comprehension. 3rd *IEEE Workshop on Program Comprehension*, Washington, Nov. 1994.

[5] B. Di Martino, B. Chapman. Program Comprehension Techniques to improve Automatic Parallelization. *Workshop on Automatic Data Layout and Performance Prediction*, Rice University, Houston TX, Apr. 1995.

[6] B. Di Martino, B. Chapman, G. Iannello, H. Zima. Integration of Program Comprehension Techniques into the Vienna Fortran Compilation System. *Int. Conf. on High Performance Computing*, New Delhi (India), Dec. 1995.

[7] M. Harandi, J. Ning. Knowledge-Based Program Analysis. *IEEE Software*, Jan. 1990.

[8] C.W. Keßler. *Automatische Parallelisierung numerischer Programme durch Mustererkennung.* PhD thesis, Universität Saarbrücken, 1994.

[9] C.W. Keßler. Symbolic Array Data Flow Analysis and Pattern Recognition in Dense Matrix Computations. *IFIP WG10.3 Working Conf. on Progr. Environments for Massively Par. Distr. Systems.* Birkhäuser, Apr. 1994.

[10] C.W. Keßler. Pattern-Driven Automatic Program Transformation and Parallelization. *3rd EUROMICRO Workshop on Parallel and Distributed Processing, San Remo.* Jan. 1995.

[11] R. Metzger. Automated Recognition of Parallel Algorithms in Scientific Applications. *Workshop on Plan Recognition at IJCAI'95*, Aug. 1995.

[12] S.S. Pinter, R.Y. Pinter. Program Optimization and Parallelization Using Idioms. *ACM SIGPLAN Principles of Programming Languages*, pp 79–92, 1991.

[13] X. Redon, P. Feautrier. Detection of Recurrences in Sequential Programs with Loops. *PARLE 93, Springer LNCS vol. 694*, pp 132–145, 1993.

[14] G. Sabot, S. Wholey. Cmax: a Fortran Translator for the Connection Machine System. *Int. ACM Conference on Supercomputing*, pp 147–156, 1993.

[15] L. Snyder. Recognition and Selection of Idioms for Code Optimization. *Acta Informatica*, 17:327–348, 1982.

[16] L.M. Wills. Automated Program Recognition: a Feasibility Demonstration. *Artificial Intelligence*, **45**, 1990.

[17] H. Zima, B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. Addison–Wesley, 1990.

[18] H. Zima, B. Chapman. Compiling for Distributed-Memory Systems. Proc. of the IEEE, Special Section on Languages and Compilers for Parallel Machines, Feb. 1993.

14
Concurrent semantics for structured design methods

P.A. Nixon and L. Shi
Department of Computer Science,
Trinity College, The University of Dublin, Dublin 2, Ireland.
Department of Computing,
Manchester Metropolitan University, Manchester, U.K.
Email: Paddy.Nixon@cs.tcd.ie, L.Shi@doc.mmu.ac.uk

Abstract
Design methods can be ambiguous due to different interpretations of symbols or concepts. This paper presents a formal semantics for the Ward/Mellor Structured Analysis Method for Real Time systems. These semantics ensures that an unambiguous meaning can be attributed to a particular design. Specifically, it ensures that concurrent and real-time properties of the design can be captured and analysed. This paper concentrates on the concurrent properties.

Keywords
Structured methods, Concurrency, Real-time systems

1 INTRODUCTION

Due to the inherent complexity of the task, the design of quality software is notoriously difficult. Worse still, the need for concurrent or real-time properties to be modelled further complicates the task, Birkinshaw et al (1994). To ease the process many design methods have been proposed which provide the software designer with notations and structures for software construction. In the real-time domain the most popular are due to Hatley/Pirbhai (1987), and Ward/Mellor (1986) , which are based on original work by DeMarco (1978). These methods aim to allow all the properties of the proposed system, including concurrency and timeliness, to be expressed in clear, unambiguous manner.

Nevertheless, for many reasons both technical and cultural, these diagramatic designs can be misinterpreted. To ensure rigorous definitions of the designers tools are made, formal semantics can be associated with the graphical methodology. This allows the designer to continue with a prefered method, whilst introducing the ability to analyse the design from a formal persepective and so removing ambiguity in the design. Work done by Elmstrøm et al, (1994) has shown the benefits of applying Petri Nets as a semantics for real-time design methods as part of the IPTES project. Yet they point out that a major

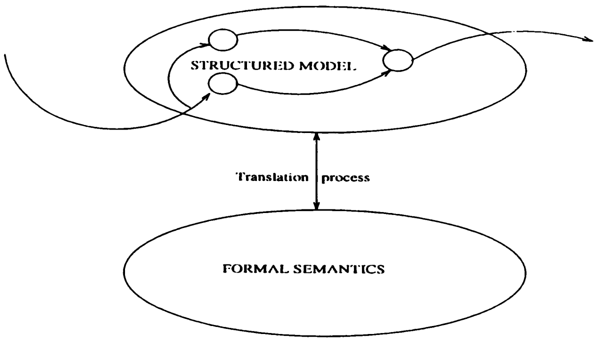


Figure 1 Notion of formal translation (Elmstrøm et al, 1994).

barrier to the widespread application of such formal semantics is that such processes can be very complex. A promising solution to the complexity problem is the use of compositional semantics. Namely, the translation process from design to formal representation is localised so that small parts of the design can be translated independently of other parts of the design. This independent translation can then lead to high speed translations and facilitate early analysis of parts of the design. This paper gives details of such a compositional approach. The paper concentrates on issues of concurrency, a complete description of the translation process is presented in Shi and Nixon (1995) and a detailed analysis of complexity for the translation is presented in Shi and Nixon (1995a)

2 BRIEF INTRODUCTION TO METHOD AND TIMED ER NETS

The particular method considered here is a the Extended Systems Modelling Language (ESML) of Rruyn et al (1988), which is a real-time extension of Tom DeMarco's structured analysis method based on data flow diagrams. Figure 2 gives the basic components of the model. The notations are defined as:

1. Transformations :
   A *data transformation* is an abstraction of a low-level system function, e.g. transforming data inputs into outputs, modifying stored data, or reporting some occurrence of event, etc..
   A *control transformation* controls the behaviour of other data/control transformations, e.g. deciding when or for how long the controlled transformations are active.
2. Data flows:
   A *discrete data flow (ddf)* is associated with a value only at discrete point of time, i.e. it is intermittently available. A *continuous data flow (cdf)* is associated with a value defined continuously over a time interval, i.e. it is continuously available.
   A *signal* is a non-value bearing data flow, it only reports that something(an event) has happened at discrete points of time.

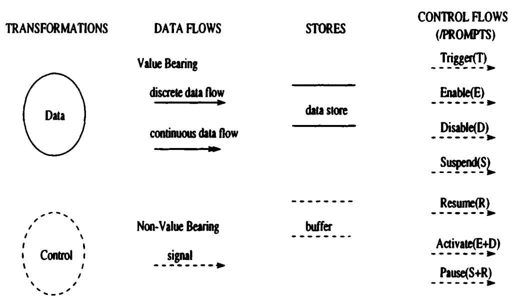


Figure 2 SA/RT notation

3. Stores:
   A *data store* acts as a repository for data that is subject to storage delay, and it is an abstraction of a file. A *buffer* actually is a special type of data store in which flows produced by transformations are subject to a storage delay before being *consumed* by other transformations. It is an abstraction of a stack or queue.
4. Control flows (or Prompts):
   Prompts represent control imposed by one control transformation on another data/control transformation, which include:

- *Trigger*: A *Trigger* causes a flow transformation to perform a time-discrete action such as producing a data flow.
- *Enable* and *Disable*: An *Enable/Disable* prompt initiates/terminates the activity of a transformation. When a transformation is disabled, it "forgets" any intermediate results and starts anew when enabled. *Activate* is a combination of *Enable* and *Disable.*
- *Suspend* and *Resume*: *Suspend* and *Resume* prompts are similar to *Enable* and *Disable*, except that a suspended transformation remembers its intermediate results and the system context and picks up where it was left off when resumed. *Pause* is a combination of *Suspend* and *Resume.*

5. Flows from multiple sources and to multiple destinations may be represented by a splitting/merging notation.

These components are then used to construct a model, or specification, which captures the data flow through the proposed system. Control of the data flow is expressed by control transformations. Concurrency is not explicitly specified in the design but is often assumed. Equally, temporal characteristics are applied in the control aspects of the design. For complete details of the method the reader is refered to Rruyn et al (1988).

2.1 Timed ER Nets

A high level net model is used here and in Elmstrøm et al (1994) to give a semantic definition to the elements of ESML, in particular capuring the temporal and concurrent aspects of the design. The specific net model used is a Timed Entity Relation net. Entitity Relation (ER) nets are Petri Nets were tokens and transitions are given different interpretations. The tokens correspond to *environments*, essentially functions that can assocaite values with variables. Transistions have *actions* associated with them, which can effect the tokens. Timed ER (TER) Nets extend the ER notation by attaching a variable *chronos* to every token, the value of which is a timestamp. This variable is modified by transitions which produce the timestamps. The usual Petri Net conventions apply to TER Nets with some small differences; the interested reader is refered to Ghezzi er al (1991). As pointed out by Ghezzi, TER nets are general enough for the requirement specification of most complex real-time systems; yet, most of the usual temporal properties are undecidable in TER nets. But generally, the TER nets can assist the analysis of specifications in the following ways:

- to restrict the analysis to special decidable subcases corresponding to special classes of applications;
- to derive approximate solutions : by ignoring token values, we reduce TER nets into low-level (timed) Petri nets. So in general, all known techniques for analyzing (timed) Petri nets can be used as approximate analysis aids in the case of TER net;
- to provide interactive decision-support systems to assess them;
- to test specifications by simulation.

Timed ER nets have been chosen as the formal model for the semantics of ESML for these reasons.

3 TRANSLATION

A translation, or formal semantic definition, of a design must capture the intentions of the designer in an unambiguous manner. Consider the example, figure 3, to translate the simple construct of storing and retrieving data from a data store. Data transformation A places data in the store C and at some point in the future data transformation B retrieves it. The definition of ESML states that the data store can be viewed as a queue, so the order of arrival of the data is important. To ensure that the design has made the correct assumptions about the data store, and not overlooked the details of the ESML definition, a formal interpretation of the design can be extracted from the original and then animated using the petri net rules. Thus, the behaviour of the design can then be relayed to the designer.

There are many possible ways of describing a given meaning for a diagram in petri nets, some more complex than others. The complexity is the key issue which must be kept to a minimum. For instance, for the data store described above a possible petri net description could included at least n places, n transitions, and n arcs to capture the n element queue which defines the buffer (where n is the size of the buffer). Alternatively, a buffer could be described by a single place, two arcs, and two transitions (see figure 4). The TER

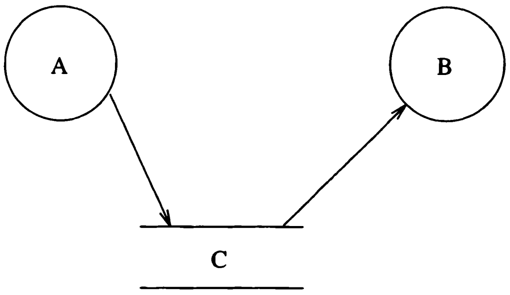


Figure 3 a simple construct

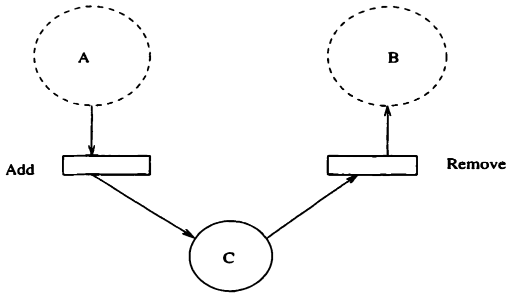


Figure 4 a simple semantics

net transitions correspond to functions. The transition for placing an element in the data store simple adds a timestamp, or chronos value, to the data stored. The transition to remove data ensures that the next data element removed is the elment with the lowest chronos value, i.e. the top of the queue.

A significant way of reducing complexity and improving efficiency is to use compositionality. This aims to modularise the semantics of the given structured method, ESML here, thus producing smaller nets to construct and analyse at a given time. The inteface between modules, or components, is well defined so that they can be combined without invalidating the work done on the indivdual components.

4 COMPOSITIONALITY

Some assumptions have to be made before any translation begins. Particularly, it has to be assumed that the SA/RT design is *complete*. By this it is meant that the usual top-down process of refining the abstract high level designs into more concrete low level refinements

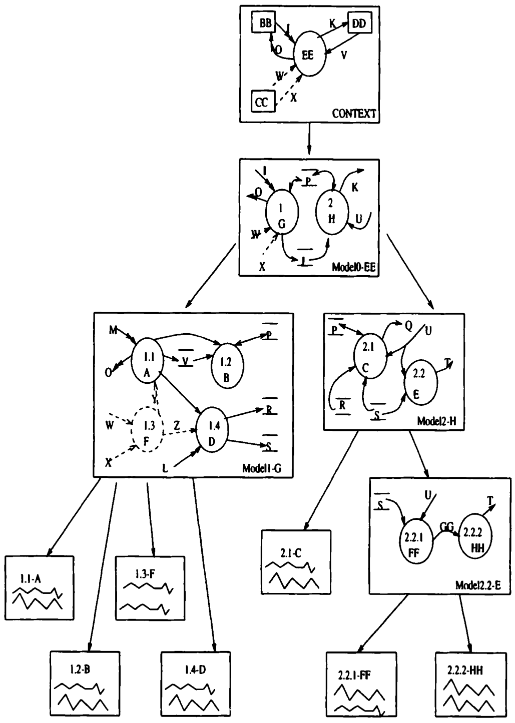


Figure 5 a hierarchical transformation schema

has taken place. Thus the translation process is dealing with a meaningful design, what ever that means. Also it is assumed that the complete specification is *flattened*. This means that the hierarchies have been removed in such a way as to leave a complete design. Figure 6 is a flattened version of Figure 5 taken from Ward and Mellor (1986).

The most important principle of the proposed translation of the is *compositionality* (or *locality*), i.e. the translation of each *component* is independent of other components. The concept of component is defined first and then the principle of compositionality is illustrated.

A *component* is either a data or control transformation together with all its inputs and outputs, or a merging or splitting structure representing a flow from multiple sources or to multiple destinations. Figure 7 illustrates a transformation schema with five components $C1$-$C5$, where $C1$-$C3$ are data transformations, $C4$ is a control transformation, and $C5$ is a merging structure which merges *ddf*1 and *ddf*2 to *ddf*3. The interface of a component includes all the data/control flows to/from it. For example, $C3$ is a data transformation with *ddf*3, *Enable* and *Disable* as input interface, and with *ddf*4, *buf* as output interface; $C5$ is a merging structure with *ddf*1 and *ddf*2 as input interface, and with *ddf*3 as output interface.

In the translation, each *component* corresponds to a TER subnet, or simply a *TER*

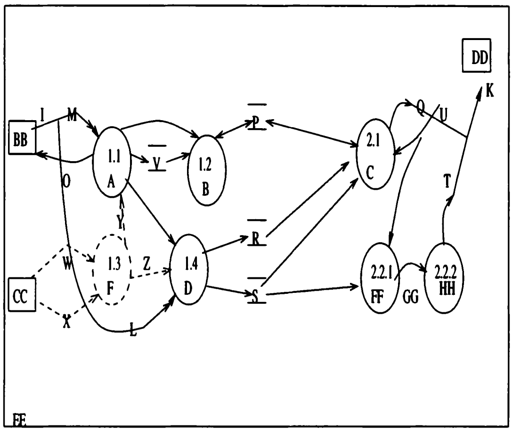


Figure 6 flattened transformation schema

net. Each flow or store connecting components in the transformation schema corresponds to a place (or some places) shared by their corresponding TER subnets. The translation rules are *compositional* (or *localized*) because each component in an SA/RT model can be translated into a TER subnet independently, and the TER net corresponding to the SA/RT model as a whole can be obtained by combining these TER subnets via shared places.

Figure 8 illustrates a TER net structure corresponding to the transformation schema TS1 in figure 7. Rectangles $C_i (1 \leq i \leq 5)$ represent TER subnets for components C_i in figure 7. Those flows in TS1 are all translated into shared places outside the rectangles. For example the discrete data flow *ddf*1 from component $C1$ to $C5$ in TS1 corresponds to a place shared by subnets $C1$ and $C5$ in the subnet for TS1. To simplify the situation, it is assumed in this example that each data/control flow corresponds to one place, the same the principle is followed when some flow corresponds to more than one place, or some group of flows share one place. Those places with no input or no output arcs, e.g. *cdf*1 and *ddf*4, correspond to flows to or from outside the schema. So the TER net as a whole is just the composition of all the five TER subnets that share interface places.

The assumption above that each flow corresponds to one place is not always true. For data flows other than data stores, two places are used for each flow as in figure 9(a,b,d); and for a control transformation, two places are used to represent all its input prompts, as illustrated in figure 9(e).

The mapping rules for flows and stores are illustrated in figure 9. For any place p, $P - cdf'$ is the complement (or empty) place of p, e.g. the token in p' in figure 9(b) denotes that no data is attached with *cdf*, in other words *cdf* is empty.

Lets consider the translation rules for *control flows* and *dataflows* in detail. Rules for other data flows in figure 9(a-c) are derived similarly and full details can be found in Shi and Nixon (1995).

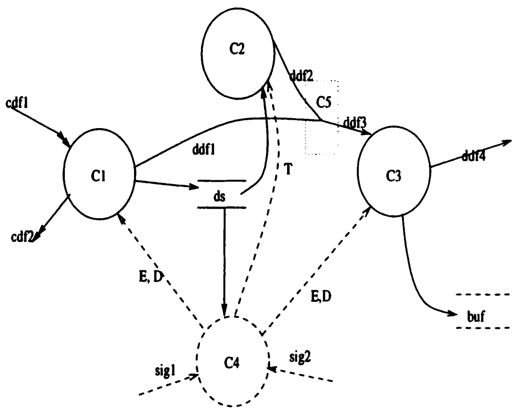


Figure 7 The concept of *component* in transformation schema TS1

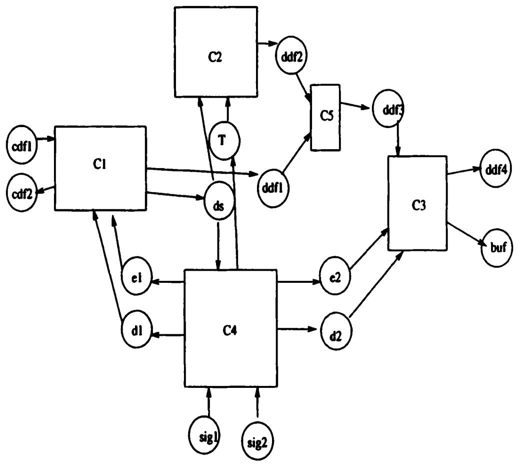
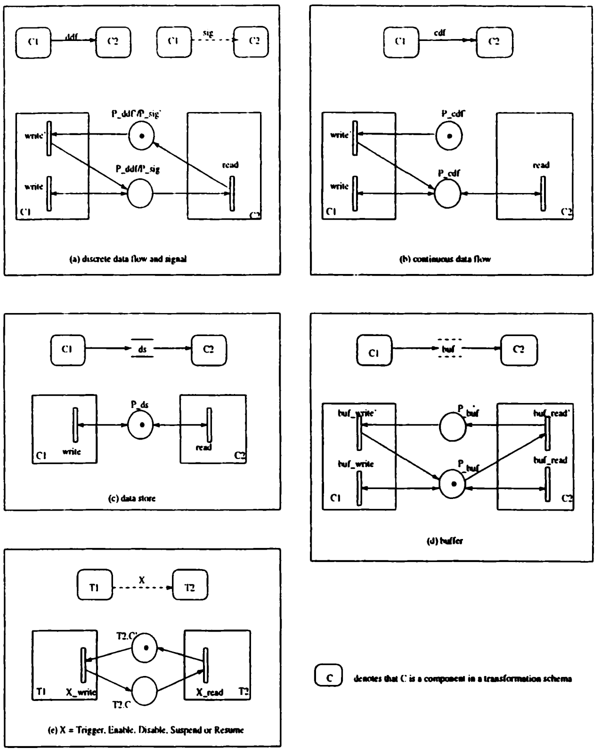


Figure 8 The compositional principle of translation for TS1



4.1 Translation of control flows

The translation in Elmstrøm et al (1994) is not compositional in that some SA/RT constructs are not translated independently. The main problem lies in the translation of control prompts, which are translated as transitions, and depend on the types, and even internal structurals of all the transformations that receive them. This problem is solved by translating all control prompts going to the same transformation as two complementary shared places, and making the translation of the controlling and controlled transformations independent of each other.

Figure 9(e) illustrates the translation method for control prompts. T1 is a control transformation which controls a (data or control) transformation T2. In the TER net, all the control prompts of T2 share two complementary places $T2.C$ and $T2.C'$. The token in $T2.C'$ indicates that no control prompt is present, and a token in $T2.C$ would carry the information of the control prompt to T2. The TER subnet for T1 has a transition X_write to produce the control prompt X; while the TER subnet for T2 has a transition X_read to consume the prompt X. But the form and number of such X_write or X_read transitions may vary, and they depend *only* on transformation T1 or T2.

Note separate places are not used here for different control prompts as for data flows.

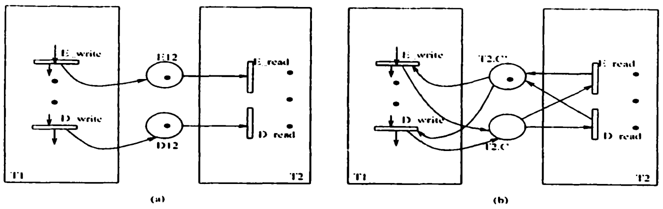


Figure 10 Translation principles for control prompts : an example

Consider the simple example as illustrated in figure 10(a), where two places $E12$ and $D12$ are used to represent the two prompts *Enable* and *Disable* sent by T1 to T2 respectively. The control prompts *Enable* and *Disable* are generated by the automaton of T1 in an orderly manner, but with the same timestamp. When *time* is not considered in the analysis, the control prompts may not be consumed in the same order as they are produced, since for transitions *E_read* and *D_read* it is indeterminate which one fires first when their preset places have the same value in *chronos*, or when *time* is not considered. But usually it is required that the prompts be consumed in the same order as they are produced. This problem can be solved by our method by using shared places for all control prompts, as in figure 10(b), where $T2.C$ is *safe* by initially putting one token in its complementary place $T2.C'$. So all the control prompts of T2 are accepted and consumed in an orderly manner by this translation.

The names, e_i, d_i, s_i, r_i and g_i (where $i = 1, 2, \ldots$) are used to indicate transitions that consume the *Enable*, *Disable*, *Suspend*, *Resume* and *Trigger* prompts respectively; arcs connected to these places can be omitted to ease reading.

4.2 Translation of Data Transformations

The next step is to consider the translation of data transformations. The concepts of *active data* and *active input* are important for understanding a data transformation. An active input arrives independently of any action of the receiving transformation, and activates the transformation when it is available (i.e. in *idle* state). An active data output is created by the activity of a data transformation, and can be an active data input for another transformation. The following definitions are assumed:

- Active data : a *ddf* not connected with data store or a signal
- Nonactive data : a *cdf* or a *ddf* connected with data store
- Active input : an active data input, or a *Trigger* prompt input *

According to the definitions above, data transformations can be placed into two classes:

1. Data transformations with an active input
   $inputs$ ::
   $active\_data\{nonactive\_data\}[Activate][Pause]$

*Note that the definition of active input in Ward (1986) does not include *Trigger*. Here we include *Trigger* to make the following description simpler.

$\quad\quad | \ Trigger\{nonactive\_data\}[Pause]$
$outputs :: \{active\_data\}^{+}\{data\_store\}$

2. Data transformations without active input
   $inputs :: \{nonactive\_data\}^{+}[Activate][Pause]$
   $outputs :: \{nonactive\_data\}^{+}\{active\_data\}$

When there is more than one active data in the output part of a data transformation, they are interpreted as alternatives; only one of them can be produced at a given time. We will consider the translation of the first of these here, translations for the other data transformations can be found in Shi and Nixon (1995).

5 CONCLUSIONS

Whatever method is used, the complexity of simulation and analysis of Petri nets grows with their size, especially the numbers of transitions and arcs. So the efficiency of simulation and analysis crucially depends on the efficiency of the translation.

The compositionality of the translation process benefits the development process of SA/RT specification models in the following aspects:

- Assisting the interactivity of the development process of SA/RT specifications
  The development of an SA/RT specification is quite an interactive process. The users modify the specification, and expect a *responsive* change in the corresponding animation and analysis. The compositional translation localizes the modification of the underlying subnets, thus improving the efficiency and interactivity.
- Assisting the incremental development of specifications
  In many occasions, the development process of a specification can be incremental. For example, a critical part of the specification may be developed and its critical properties need to be analyzed first. The Compositional translation allows the translation and analysis of part of the model, thus supporting the incremental development of specifications.
- Assisting modular development and analysis of specifications
  Any module in transformation schema can be translated independently into a Petri net module; Petri net modules can be combined just by shared places. Modifying any part of the specification only results in *localized* modifications of the underlying net and other parts, including their properties, will be kept intact. Thus the compositional translation is essential to the compositional/modular development and analysis of specifications.

In summary, the compositional and efficient translation given here can benefit the analysis and the development of SA/RT specifications. The work has also highlighted the importance of the modularity/compositionality of TER nets.

Future work will include investigation into the compositionality of temporal properties and their transformation into HLTPNs. Work is also beginning on a prototype tool for automating the process presented. On a more theoretical note, our research would benefit from more work on the efficient analysis of high-level timed Petri nets and especially their modular/compositional analysis. finally, the authors believe the translation is not tied to the SA/RT described in the paper and work is proceeding to substantiate this hypothesis.

6 REFERENCES

W. Rruyn, R. Jensen, D. Keskar, and P. Ward. (1988) *ESML : An extended systems modeling language based on the data flow diagram.* ACM SIGSOFT, Software Engineering Notes, 13(1):58–67, Jan. 1988.

C. Ghezzi, D. Mandrioli, S. Marasca, and M. Pezzé. (1991) *A unified high-level Petri net formalism for time-critical systems.* IEEE SE, 17(2):160–172, Feb. 1991.

C I Birkinshaw P R Croll D G Marriott and P A Nixon, (1994) *Engineering safety-critical parallel systems* , In Information and Software Technology, Vol. 36, No. 7, pp. 449-456, 1994.

R. Elmstrøm, R. Lintulampi, and M. Pezzé. (1994) *Automatic translation of SA/RT to high-level timed Petri nets.* Technical Report IPTES-PDM-17-V2.3, Odense, Jan 1994.

T. DeMarco. (1978) *Structured Analysis and System Specification* . New Jersey, Prence-Hall.

D. J. Hatley and I. A. Pirbhai. (1987) *Strategies for Real Time Specifications.* New York, Dorset House.

P. Ward and S. Mellor. (1985) *Structured Development for Real-Time Systems*, volume 1-3. New Jersey: Prence Hall.

Paul T. Ward. (1986) *The transformation schema: an extension of the data flow diagram to represent control and timing.* IEEE SE, 12(2):198–210, Feb 1986.

L. Shi and P. Nixon. (1995a) *An improved translation from SA/RT model to high-level timed Petri nets.* In the proceedings of FME'96, to appear in LNCS, Springer Verlag.

L. Shi and P. Nixon. (1995) *Uniting formal and structured design methods for real-time systems.* Technical report, Dept. of Computing, Manchester Metropolitan University, 1995.

7 BIOGRAPHY

Patrick Nixon recieved his B.Sc in Computer Science from Liverpool University in 1990 and his Ph.D in Computer Science from Sheffield University in 1994. He was a lecturer in Computing at Manchester Metropolitan University until 1995, when he took up a post of lecturer at Trinity College Dublin. His research interests include software engineering, parallel and distributed computing, distributed object systems, and applied formal methods.

Lihua Shi is currently a Ph.D student in Computing at Manchester Metropolitan University. Prior to this she recieved her B.Sc in Computer Science and a Masters in Software Engineering from East China Normal University, and was a lecturer there until 1994.

ACKNOWLEDGEMENTS

This research is supported by a research scholarship from Manchester Metropolitan University for Lihua Shi.

15
Towards a theory of shared data in distributed systems

S. Dobson and C.P. Wadsworth
Rutherford Appleton Laboratory
Chilton, Didcot, Oxfordshire OX11 0QX, UK
Tel +44 1235 445867 Fax +44 1235 445945
E-mail {S.Dobson, C.P.Wadsworth}@ rl.ac.uk

Abstract
We have developed a *theory of sharing* which captures the behaviour of programs with respect to shared data into the framework of process algebra. The core theory can describe programs performing read and write access to unitary pieces of shared data. Extensions allow shared data to be decomposed and atomic copies to be made, reflecting the common operations of parallel programs. We describe the theory and give an example of its use in analysing and transforming a sample mathematical application.

Keywords
Sharing, process algebra, program analysis, program transformation

1 INTRODUCTION

Multiprocessor systems traditionally fall into two camps: shared memory, in which all processes have direct access to all the data in a computation; and distributed memory, in which access to much of the data involves explicit communication. Recently the distinction has become a little blurred through the use of virtual shared memory (Frank, 1992)(Li, 1989) and through the desire to make use of high-level data abstractions when building distributed applications.

In a typical distributed application there will be a body of data which is shared between some or all components. Examples include a mesh in a simulation, a set of tables in a distributed database, or a collection of pages in a distributed multimedia application. The efficient implementation of the application may require exploration of several different strategies before the "best" is chosen.

The danger is that decomposition blurs the programmer's conceptual model of the data being manipulated, and introduces subtle machine dependencies. These can make large applications hard to analyse and maintain, and damage their portability.

As part of the TallShiP collaborative project between RAL and the University of Leeds we have been investigating the use of high-level typed data abstractions for creating distributed applications. Our contention is that this approach raises the level of distributed programming so as to allow a more structured, more portable and more easily-analysed – in short, better engineered – applications to be created. At the same time, it allows us to exploit type-specific information for optimised processing and distribution of the components of an application.

In order to explore the ways in which applications share data, we are developing a *theory of sharing*. The intention of this theory is to capture the sharing behaviour of applications, allowing different patterns of sharing to be identified, characterised and compared. We hope that this will lead to new insights into the design of efficient, portable shared data types, and to new methods for the optimised compilation and support of distributed applications.

Section two introduces the core theory of sharing. Section three describes two extensions covering wider class of systems. Section four presents some preliminary applications of the theory to program analysis. Section five relates the theory to similar work, and section six offers our conclusions and some directions for future work.

2 CORE THEORY OF SHARING

Sharing Areas and Events

Many kinds of data may be shared in an application, from simple variables to large structured types or multimedia objects. For our purposes, however, all shared data is represented by the single abstraction of a *sharing area*. A sharing area is a collection of zero or more named data items. A single variable is represented by a sharing area having one element; a large array by an area with many elements each identified by an index tuple. For the time being we consider sharing areas to be without internal structure; they are also *untyped* in that the theory does not describe the contents of elements or how they are named. All sharing areas are disjoint from all other sharing areas, in that no two areas share elements in common.

Having defined an abstraction for shared data we need ways in which to access it. Suppose that we have a set `S` of sharing areas, denoted `a,b,...`. A procedure may atomically read zero or more elements from a single sharing area: this action is denoted by `rd(a)` where `a` is the sharing area accessed. Similarly the action `wr(a)` denotes the atomic update of elements in sharing area `a`. We term these two basic actions the*events* of sharing theory.

Sharing Expressions

A *sharing expression* describes a program's interactions with sharing areas, built by composing the basic actions which may be applied to areas. Each sharing expression is a term in a modified process algebra (based on the system PA of Baeten and Weijland (1990)) using events instead of communication as the basic elements.

Events may be built into larger expressions using sequential composition, alternative composition and parallel composition, denoted by the functions `;`, `+` and ‖ respectively*. So the term `rd(a);(wr(a)+(wr(a)‖wr(b)))` describes the sharing behaviour of a function which first reads elements from sharing area `a` and then *either* updates elements in `a` *or* updates elements in `a` and `b` in parallel. In PA, as in most process algebras, parallel composition is viewed as non-deterministic interleaving.

* PA uses `.` (dot) rather than `;` (semi-colon) to denote sequential composition.

We may define a set of equations on the terms created from the events and combining operators. If x, y, z represent arbitrary terms, then:

$$x+y = y+x \qquad (x+y)+z = x+(y+z) \qquad x+x = x$$
$$(x+y);z = x;z+y;z \qquad (x;y);z = x;(y;z) \qquad x \| y = y \| x$$

New sharing areas are introduced using the υ operator: the expression $\upsilon a.x$ (where x is an arbitrary term) introduces a new sharing area a for use in x (called the *scope* of a). For simplicity we assume that no sharing area name is ever re-used.

A sharing area a is said to *occur* in a term x if x contains an event $rd(a)$ or $wr(a)$. Events in a occurring inside the scope of a υa operator appear *bound*; any other occurrences appear *free*. We define using structural induction a function FA which computes the set of sharing areas appearing free in a term:

$$FA(rd(a)) = a \qquad FA(x;y) = FA(x) \cup FA(y) \qquad FA(x+y) = FA(x) \cup FA(y)$$
$$FA(wr(a)) = a \qquad FA(x \| y) = FA(x) \cup FA(y) \qquad FA(\upsilon a.x) = FA(x) \setminus \{a\}$$

We then use FA to define side conditions for equations relating to the υ operator, controlling the scope of sharing areas in terms:

$$\upsilon a.(x;y) = x;\upsilon a.y \quad a \notin FA(x) \qquad \upsilon a.(x;y) = (\upsilon a.x);y \quad a \notin FA(y)$$
$$\upsilon a.(\upsilon b.x) = \upsilon b.(\upsilon a.x) \qquad \upsilon a.x = x \quad a \notin FA(x)$$
$$(\upsilon a.x)+(\upsilon b.y) = \upsilon a.\upsilon b.(x+y)$$
$$a \notin FA(y) \wedge b \notin FA(x)$$

The first two equations state that terms which do not actually use a sharing area may be moved into or out of its scope. The fourth axiom states that unused sharing areas may be deleted, or conversely that new areas may be introduced freely. The final axiom allows sharing areas to be factored into or out of alternative compositions.

Renaming Operators

PA defines a small set of renaming operators, allowing actions to be changed into other actions. We may usefully import this idea into sharing theory.

A *renaming function* is a function which maps actions to actions. The renaming function $r(f)$ maps every action f to some other action g (which may be the same as f). A renaming function ρ_r applies r to a term, and is defined by a straight-forward structural induction. So in the term $x = \upsilon a.\upsilon b.(rd(a);wr(b))$ if we define $r = id\{wr(b) \mapsto wr(a)\}$ then $\rho_r(x) = \upsilon a.\upsilon b.(rd(a);wr(a))$ changes every $wr(b)$ action into a $wr(a)$ action.

Note that ρ_r changes actions, not sharing areas. We may however use it to define another operator which renames sharing areas in a term. Let x be a term. Let S be a sub-set of the sharing areas occurring in x, and let S' be a set of new sharing areas not occurring in x. Let s be a *sharing area renaming function* mapping elements of S to elements of S'. Now create a renaming function such that for all $a \in S$ and $b=s(a)$ we have $r(rd(a)) = rd(b)$, $r(wr(a)) = wr(b)$ and r is the identity on all other actions. We now define a *sharing area renaming operator* α_s which renames sharing areas and their events. We first introduce each new sharing area in S' using the υ

operator, then rename according to the renaming function induced by $\mathtt{s}$. Using our example term $\mathtt{x}$ from above, if we set $\mathtt{s} = \mathtt{id\{b\mapsto c\}}$ then

$$\begin{array}{llr}
\alpha_s(\mathtt{x}) & = \alpha_s(\upsilon\mathtt{a}.\upsilon\mathtt{b}.(\mathtt{rd(a);wr(b)})) & \\
& = \upsilon\mathtt{c}.\upsilon\mathtt{a}.\upsilon\mathtt{b}.(\rho_r(\mathtt{rd(a);wr(b)})) & \text{for } \mathtt{c}\notin \mathtt{FA(x)} \\
& = \upsilon\mathtt{c}.\upsilon\mathtt{a}.\upsilon\mathtt{b}.(\mathtt{rd(a);wr(c)}) & \\
& = \upsilon\mathtt{c}.\upsilon\mathtt{a}.(\mathtt{rd(a);wr(c)}) & \text{eliminating unused area } \mathtt{b}
\end{array}$$

The usefulness of this operator will become apparent in the example (section 5).

Effect Analysis

We define a function $\mathtt{AE}$ to extract the free sharing areas in which the term causes events. The function generates two sets, containing the areas in which read events occur and the areas in which write events occur:

$$\begin{array}{ll}
\mathtt{AE(rd(a))} = (\{\mathtt{a}\},\varnothing) & \mathtt{AE(wr(a))} = (\varnothing,\{\mathtt{a}\}) \\
\mathtt{AE(x+y)} = \mathtt{AE(x)+AE(y)} & \mathtt{AE(x;y)} = \mathtt{AE(x)+AE(y)} \\
\mathtt{AE(x\|y)} = \mathtt{AE(x)+AE(y)} & \mathtt{AE(\upsilon a.x)} = \mathtt{AE(x)}-(\{\mathtt{a}\},\{\mathtt{a}\})
\end{array}$$

(where + and - respectively denote pointwise set union and set difference on pairs). We use two functions $\mathtt{AE}_{\mathtt{rd}}$ and $\mathtt{AE}_{\mathtt{wr}}$ to project the first and second sets, so $\mathtt{AE}_{\mathtt{rd}}\mathtt{(x)}$ is the set of sharing areas in which $\mathtt{x}$ causes read events.

3 EXTENSIONS

The core theory can express sharing in an important class of algorithms, but lacks some of the features commonly encountered in parallel and distributed applications. We shall now extend it in two directions, to encompass the decomposition and copying of data structures. These extensions are completely modular, in that they may be added individually or together to the core theory to generate a more expressive system.

Sub-Areas

Expressing algorithms as manipulations on large shared types can make applications more analysable. However, direct implementation of a large value as a single object may lead to contention and centralisation which would damage performance. Programmers must often decompose a data structure (such as a grid) into sub-parts which are then distributed and processed in parallel. There will in general be many different decomposition strategies for a value, each semantically equivalent but with different performance profiles. We would like to capture the decomposition of a value without losing the fact that the sub-parts combine to form a single, larger whole.

Disjointness, Containment and Decomposition

Two sharing areas $\mathtt{a}$ and $\mathtt{b}$ are *disjoint* (denoted $\mathtt{a}\oplus\mathtt{b}$) do not share elements in common. In the core theory, all sharing areas are mutually disjoint. We may relax this restriction and allow two areas share some elements. Of particular interest is the case where all the elements of a sharing area b are also elements of an area a, so that b identifies a sub-part of a. We denote this by $b \subset a$, and say that a *contains* b (or that b is a *sub-area* of a).

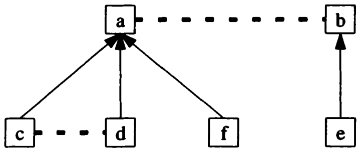


Figure 1 Sub-areas form a tree under the containment relation.

Introducing sub-areas into the theory of sharing requires another operator. If a is a sharing area, the term $\Delta_a\{a_1, a_2, \ldots, a_n\}.x$ introduces the set of disjoint sub-areas $a_1, a_2, \ldots, a_n$ of a into x. Each a_i is a sub-area of a, so $a_i \subset a$. There may be elements of a not contained in any a_i. Both the υ and Δ operators introduce sharing areas. However, υ generates new (shared) state whereas Δ simply partitions existing state. So the term

$$\upsilon a.\upsilon b.(x;(\Delta_a\{c,d\}.\Delta_b e.(y;(\Delta_a f.z))))$$

describes a set of sharing areas which form a forest of trees (figure **1**, where the arrows denote containment of one area within another and the dotted lines denote areas which are explicitly known to be disjoint). We may also assert that if $a \oplus b$, $c \subset a$ and $e \subset b$ then $c \oplus e$ (and similarly for d); but it is not necessarily the case that $c \oplus f$. Extending the definition of FA to encompass this new term

$$FA(\Delta_a D.x) = FA(x) \setminus D$$

for some set of sub-areas D, we may define some additional equations for the extended theory of sharing with sub-areas:

$$\begin{array}{ll}
\Delta_a D.(x;y) = (\Delta_a D.x);y & D \cap FA(y) = \varnothing \\
\Delta_a D.(x;y) = x;(\Delta_a D.y) & D \cap FA(x) = \varnothing \\
\Delta_a D.(\Delta_b E.x) = \Delta_b E.(\Delta_a D.x) & \\
\Delta_a \varnothing.x = x & \\
\Delta_a (D \cup d).x = \Delta_a D.x & d \notin FA(x) \\
(\Delta_a D.x) + (\Delta_b E.y) = \Delta_a D.\Delta_b E.(x+y) & D \cap FA(y) = \varnothing \wedge E \cap FA(x) = \varnothing
\end{array}$$

Note that there is no equation of the form $\Delta_a D.\Delta_a E.x = \Delta_a(D \cup E).x$ as disjointness of sub-areas is not implied across separate Δ terms.

Generalisation

It often impedes the understanding of an application if the partitioning of data structures is made too explicit, and this is also true of sharing theory: the use of sub-areas can make analysis unnecessarily difficult. The *generalisation transform* allows us to abstract away from different sub-area decompositions where necessary.

If $\mathtt{a}$, $\mathtt{a}_1$ and $\mathtt{a}_2$ are sharing areas such that $\mathtt{a}_1, \mathtt{a}_2 \subset \mathtt{a}$, we say that $\mathtt{a}$ is a *generalised sharing area* of $\mathtt{a}_1$ and $\mathtt{a}_2$. ($\mathtt{a}$ may itself be a sub-area of another, larger area.) Let $\mathtt{S}$ be a set of mutually disjoint sharing areas, and let $\mathtt{r}$ be a renaming function. The generalisation transform $\Gamma_{\mathtt{r}}{}^{\mathtt{S}}$ is defined by another structural induction, the only interesting case of which is

$$\Gamma_r{}^S(\Delta_a D.x) = \begin{cases} \Delta_{\mathbf{a}} D.\Gamma_r{}^S(x) & \text{if } \neg\exists c \in S.a \subset c \\ \Gamma_{r'}{}^S(x) \text{ where } r' = r\{rd(a_i) \mapsto rd(c), wr(a_i) \mapsto wr(c) | a_i \in D\} & \text{if } \exists c \in S.a \subset c \end{cases}$$

In a $\Delta_{\mathtt{a}}$ term, if there is an area in $\mathtt{S}$ which is a generalisation of $\mathtt{a}$ (there can be at most one such area) then the events in the scope of the term are re-written so as to be events in the generalised area.

The generalisation transform can generalise a selection of sharing areas to abstract away from any decomposition, or can eliminate decomposition entirely. Many terms may have the same generalisation, so generalisation may be seen as a form of refinement: two terms $\mathtt{y}$ and $\mathtt{y'}$ such that $\Gamma_{\mathtt{id}}{}^{\mathtt{S}}(\mathtt{y}) = \Gamma_{\mathtt{id}}{}^{\mathtt{S}}(\mathtt{y'})$ for some $\mathtt{S}$ can be considered to be "the same" *modulo* different decompositions.

Copy Events

Suppose we have a sharing area representing some commonly-read state in an application. A useful optimisation might be to pre-copy the data to sites which use it, forming a local cached copy at each site. This would reduce the number of accesses to a single point in the system.

To capture this notion within sharing theory, we introduce a new set of actions. For every pair $\mathtt{a,b}$ of sharing areas we define an event $\mathtt{cp(a,b)}$ denoting the atomic "snapshot" of all the elements of $\mathtt{a}$ into $\mathtt{b}$. Immediately after a copy event $\mathtt{b}$ is identical to $\mathtt{a}$. The effect of performing some action which uses $\mathtt{a}$ is the same as performing the same action using an identical copy of $\mathtt{a}$:

$$\upsilon \mathtt{a.x} = \upsilon \mathtt{a}.\upsilon \mathtt{b}.(\mathtt{cp(a,b)};\alpha_{\{\mathtt{a}\mapsto \mathtt{b}\}}(\mathtt{x})) \qquad \mathtt{b} \notin \mathrm{FA}(\mathtt{x})$$

In the presence of sub-areas we refine the definition slightly to prevent pathological cases such as copying the contents of an area into one of its sub-areas. We do this by restricting the existence of $\mathtt{cp(a,b)}$ events to those cases where $\mathtt{a} \oplus \mathtt{b}$. Having done this, we may define the behaviour of copy actions on sub-areas as

$$\Delta_{\mathtt{a}}(\mathtt{D} \cup \mathtt{d}).\mathtt{x} = \Delta_{\mathtt{a}}(\mathtt{D} \cup \mathtt{d}).\upsilon \mathtt{c}.(\mathtt{cp(d,c)};\alpha_{\{\mathtt{d}\mapsto \mathtt{c}\}}(\mathtt{x})) \qquad \mathtt{c} \notin \mathrm{FA}(\mathtt{x})$$

where $\mathtt{d}$ is some sub-area of $\mathtt{a}$.

4 APPLICATIONS

The theory we have presented above allows us to capture the sharing behaviour of a wide class of programs. The programs may then be analysed for side effects, possible conflicts or non-determinism, and potentialoptimisations.

Program Analysis

A function or procedure within a program gives rise to a sharing expression which captures its dependence and influence on shared state. One may define a mapping from a language to sharing theory, and then manipulate the sharing expression to draw conclusions about the code. Our aim is to also reverse this mapping, to use sharing theory as the basis of a transformation system – to date we have concentrated on the analysis phase.

Conflicts and Synchronisation

Suppose we have a program in which an object is being updated in parallel by two functions. We represent the object by a sharing area a, and the two functions x and y as having behaviour given by $rd(a);wr(a);...;wr(a)$. The overall behaviour of this system is given by the expression $z = \nu a.(x \| y)$. If we compute AE for x and y, we discover that $a \in AE_{wr}$ for both. This indicates that the parallel term has a potential conflict as different interleavings may introduce write events in different orders, making the program non-deterministic.

We say that two terms *interact* if the events caused by one may affect the behaviour of the other in terms of the results of read events. Two terms *interfere* if they interact or if they cause write events in a common area. Non-interacting terms cannot directly affect each others' behaviour as they do not update any state accessed by the other. Interference is a stronger condition which also encompasses terms which, while not necessarily affecting each other's actions, may still generate non-deterministic final effects if composed in parallel. Both interaction and interference are properties which may simply be determined by examination of the terms involved.

In some cases interference is observed "spuriously" because sharing theory works at the level of sharing areas, not elements within those areas. For example if the sharing area represents a large grid and the interfering functions are updating different parts of it, then there is no problem. If this property is captured by means of sub-areas, the interference is removed. Structured algorithm or type design can help make this information available.

For other cases, interference constrains an application to ensure that the atomicity of events is maintained, using locking *et cetera*. The converse is also true: in the absence of interference, no concurrency control is needed. This means that the theory can detect cases in which concurrency control may be "switched off" to avoid overheads.

Caching and Copying

Many algorithms make many more read accesses to shared data than write accesses, and it may be advantageous to create cached copies of the shared state local to each process rather than have all processes share a single copy. Sharing theory may be used to detect situations in which caching may be applied. The basic technique is to observe parts of a term in which a sharing area incurs only read events in parallel, and then create new copies of the area local to each parallel process.

For example, let x and y describe functions which repeatedly read elements from a sharing area a in order to update an area b, so that no write events are caused in a. We may transform this expression to introduce private copies ofa:

$$\begin{aligned} X &= \nu a.\nu b.(x \| y) \\ &= \nu a.\nu b.((\nu c.x) \| (\nu d.y)) \\ &= \nu a.\nu b.((\nu c.(cp(a,c);\alpha_{(a\mapsto c)}(x))) \| (\nu d.(cp(a,d);\alpha_{(a\mapsto d)}(y)))) \end{aligned}$$

In a distributed system such caching may be highly advantageous. Rather than access a single copy of some shared data, possibly involving network access, it is possible to generate a term which uses local copies of the data and is provably equivalent to the shared-data case. The technique is particularly effective in the presence of sub-areas, where we may generate local copies of only those parts of a piece of shared state which are actually needed in each partial computation.

However, not all such opportunities for caching and replication will be equally advantageous. There is a hidden assumption – not always made explicit in work on transformation – that accessing local copies is far less expensive than accessing a shared copy and justifies the copying overhead. In systems which make highly infrequent access to shared state it may not be worth performing this optimisation. Deciding between these two situations is an interesting problem.

Example: Parallel Solution of the 2-D Wave Equation

To demonstrate these techniques we shall analyse a program calculating the numerical solution of a partial differential equation. The application models the motion of a wave in a fluid medium – for example a pressure wave in a gas. In two dimensions this equation has the discrete form

$$C[i,j] = B[i,j] - A[i,j] + \frac{1}{4}(B[i+1,j] + B[i-1,j] + B[i,j+1] + B[i,j-1])$$

where three grids *A*, *B* and *C* are used to hold values of the simulation at different time steps. Grid *B* holds the values of points at time *t*; grid *A* at time *t-1*; and grid *C* holds the new values for time *t+1*. The new value of a point *(i, j)* is computed as a function of its past value and those of its immediate four neighbours. Let us assign sharing areas `a`, `b` and `c` to represent the grids *A*, *B* and *C* respectively. These sharing areas contain many elements, one for each point in the grids.

Simple Sharing Analysis

To compute the new value of a point we apply a function `NewValue` which accesses areas `a` and `b` in order to compute a value with which to update area `c`. This gives rise to the sharing expression:

```
newvalue  = (rd(b)‖rd(a)‖rd(b)‖rd(b)‖rd(b)‖rd(b));wr(c)
```

Calculation of a single time-step involves applying `NewValue` to each point in the space in parallel:

```
calc = newvalue‖newvalue‖...‖newvalue
```

where each instance has the same sharing behaviour, but updates a different point in `c`.

Every instance of `newvalue` in `calc` interferes with every other instance, as each is updating `c`. However, as we know from the structure of the calculation that each instance updates a different point, this interference is spurious. We may make the structure explicit by decomposing `c` into sub-areas $c_1, c_2, \ldots, c_n$ such that each sub-area contains a single point. If we then apply each instance of `newvalue` to a different sub-area:

$$\texttt{calc}' = \Delta_c\{c_1, c_2, \ldots, c_n\}.\ \alpha_{(c\mapsto c1)}(\texttt{newvalue}) \| \alpha_{(c\mapsto c2)}(\texttt{newvalue}) \| \ldots \| \alpha_{(c\mapsto cn)}(\texttt{newvalue})$$

the spurious interference has disappeared – at the price of an extremely complex sub-area structure. However, note that `calc` = Γ_r^S`(calc')`: `calc'` is simply a refinement of `calc` using a particular decomposition strategy, and we can easily generalise it to retrieve the simpler form.

Another possible strategy is to divide the decomposed parts of `c` into (say) four sets for distribution onto four processors:

```
calcd = Δc{p,q,r,s}.
            (Δp{p1,p2,...,pn}.
                    (α{c↦p1}(newvalue)‖α{c↦p2}(newvalue)‖...))
           ‖(Δq{q1,q2,...,qm}.
                    (α{c↦q1}(newvalue)‖α{c↦q2}(newvalue)‖...))
           ‖...
```

In this case we see that the "processor" terms are non-interfering, and the instances of `newvalue` within each term are also non-interfering. Furthermore we may generalise the decomposition of the processor terms to obtain a term describing the events at each processor, and then further generalise to obtain `calc`.

Copying and Storage Re-use

Inspecting the discrete form of the wave equation, we see that the grids *A*, *B* and *C* are used cyclically: the values at time*t* become those at time*t-1* on the next cycle of computation. Abstractly

- the system calculates the values of*C* using those of*A* and *B*;
- it then creates three new grids*A'*, *B'* and *C'*;
- it copies the values of*B* into*A'* and *C* into*B'*; and
- it then performs the next cycle of calculation using the new grids.

This behaviour is captured by the expression

```
νa'.νb'.νc'.((cp(b,a')‖cp(c,b'))...)
```

and we may use this simple description of the system's behaviour and derive a new expression which performs "pointer swapping" and re-uses the existing storage without copying.

Let us consider the full definition of two cycles of computation, using the basic definition of `calc` for simplicity. The calculation has the behaviour:

```
wave = νa.νb.νc.(setup;twostep)
setup = wr(a)‖wr(b)
twostep = calc;calc2
calc2 = νa'.νb'.νc'.(cp(b,a');cp(c,b');α{a↦a',b↦b',c↦c'}(calc))
```

(where `setup` initialises the grids *A* and *B* with the initial state of the system). So the computation initialises the grids, calculates the first iteration, creates new grids and initialises them by copying, and then repeats the calculation. The optimised version replaces `twostep` with:

```
twostep' = calc;α{a↦b,b↦c,c↦a}(calc)
```

where the grids from the first step are re-assigned in the second.
Optimising `wave` involves converting `twostep` into `twostep'`. The only observation which we need to make involves the use of a copy event as the final action on a sharing area. If a copy is made of an area, and the original is never accessed again, then one may equally make use of the original area instead of the copy. This is expressed by the equation:

$$\upsilon\mathtt{a}.(\mathtt{x};\upsilon\mathtt{b}.(\mathtt{cp}(\mathtt{a},\mathtt{b}).\mathtt{y})) = \upsilon\mathtt{a}.(\mathtt{x};\alpha_{\{\mathtt{b}\mapsto\mathtt{a}\}}(\mathtt{y})) \qquad \mathtt{a}\in \mathtt{FA}(\mathtt{y})$$

Using this equation, we may transform`twostep` as follows:

$$\begin{aligned}
\mathtt{twostep} &= \mathtt{calc};\mathtt{calc2}\\
&= \mathtt{calc};\upsilon\mathtt{a}'.\upsilon\mathtt{b}'.\upsilon\mathtt{c}'.(\mathtt{cp}(\mathtt{b},\mathtt{a}');\mathtt{cp}(\mathtt{c},\mathtt{b}');\\
&\qquad\qquad \alpha_{\{\mathtt{a}\mapsto\mathtt{a}',\mathtt{b}\mapsto\mathtt{b}',\mathtt{c}\mapsto\mathtt{c}'\}}(\mathtt{calc}))\\
&= \mathtt{calc};\upsilon\mathtt{a}'.\upsilon\mathtt{c}'.(\mathtt{cp}(\mathtt{b},\mathtt{a}');\upsilon\mathtt{b}'.(\mathtt{cp}(\mathtt{c},\mathtt{b}');\\
&\qquad\qquad \alpha_{\{\mathtt{a}\mapsto\mathtt{a}',\mathtt{b}\mapsto\mathtt{b}',\mathtt{c}\mapsto\mathtt{c}'\}}(\mathtt{calc})))\\
&= \mathtt{calc};\upsilon\mathtt{a}'.\upsilon\mathtt{c}'.(\mathtt{cp}(\mathtt{b},\mathtt{a}');\alpha_{\{\mathtt{b}'\mapsto\mathtt{c}\}}(\alpha_{\{\mathtt{a}\mapsto\mathtt{a}',\mathtt{b}\mapsto\mathtt{b}',\mathtt{c}\mapsto\mathtt{c}'\}}(\mathtt{calc})))\\
&= \mathtt{calc};\upsilon\mathtt{c}'.\upsilon\mathtt{a}'.(\mathtt{cp}(\mathtt{b},\mathtt{a}');\alpha_{\{\mathtt{b}'\mapsto\mathtt{c}\}}(\alpha_{\{\mathtt{a}\mapsto\mathtt{a}',\mathtt{b}\mapsto\mathtt{b}',\mathtt{c}\mapsto\mathtt{c}'\}}(\mathtt{calc})))\\
&= \mathtt{calc};\upsilon\mathtt{c}'.(\alpha_{\{\mathtt{a}'\mapsto\mathtt{b}\}}(\alpha_{\{\mathtt{b}'\mapsto\mathtt{c}\}}(\alpha_{\{\mathtt{a}\mapsto\mathtt{a}',\mathtt{b}\mapsto\mathtt{b}',\mathtt{c}\mapsto\mathtt{c}'\}}(\mathtt{calc}))))\\
&= \mathtt{calc};\upsilon\mathtt{c}'.(\alpha_{\{\mathtt{a}\mapsto\mathtt{b},\mathtt{b}\mapsto\mathtt{c},\mathtt{c}\mapsto\mathtt{c}'\}}(\mathtt{calc}))
\end{aligned}$$

which eliminates the intermediate grids *A'* and *B'*, but still generates a new grid *C'* for each cycle. We may re-use the grid *A* instead of creating *C'*, but the system has no way of determining this: if we indicate it, by using an additional copy action, then

$$\begin{aligned}
\mathtt{twostep} &= \mathtt{calc};\upsilon\mathtt{c}'.(\mathtt{cp}(\mathtt{a},\mathtt{c}');\alpha_{\{\mathtt{a}\mapsto\mathtt{b},\mathtt{b}\mapsto\mathtt{c},\mathtt{c}\mapsto\mathtt{c}'\}}(\mathtt{calc}))\\
&= \mathtt{calc};\alpha_{\{\mathtt{c}'\mapsto\mathtt{a}\}}(\alpha_{\{\mathtt{a}\mapsto\mathtt{b},\mathtt{b}\mapsto\mathtt{c},\mathtt{c}\mapsto\mathtt{c}'\}}(\mathtt{calc}))\\
&= \mathtt{calc};\alpha_{\{\mathtt{a}\mapsto\mathtt{b},\mathtt{b}\mapsto\mathtt{c},\mathtt{c}\mapsto\mathtt{a}\}}(\mathtt{calc})
\end{aligned}$$

and we have achieved our aim.

Halos and Caching

The method described above is a generic technique, applicable to any system based around iterative time-series methods. The most common implementation of such divides the grids into disjoint regions which are mapped onto different processors for calculation in parallel, often with "halos" of data dependencies between regions.

Let us define a distributed memory two-step solver on four processors using a quadrant partitioning. Computing a point in a particular region of `C` may involve access to points in the corresponding and neighbouring regions of `A` and `B`. The distributed update function for a particular region, $\mathtt{newvaluer}_n$, is thus defined by

$$\begin{aligned}
\mathtt{newvaluer}_n &= (\mathtt{rd}(\mathtt{b}_n)\,\|\,\mathtt{rd}(\mathtt{a}_n)\,\|\,\mathtt{rdhalo}_n);\mathtt{wr}(\mathtt{c}_n)\\
\mathtt{rdhalo}_n &= (\mathtt{rd}(\mathtt{b}_n)+\mathtt{rd}(\mathtt{b}_{n\text{-}\mathtt{left}}))\,\|\,(\mathtt{rd}(\mathtt{b}_n)+\mathtt{rd}(\mathtt{b}_{n\text{-}\mathtt{right}}))\,\|\\
&\qquad (\mathtt{rd}(\mathtt{b}_n)+\mathtt{rd}(\mathtt{b}_{n\text{-}\mathtt{up}}))\,\|\,(\mathtt{rd}(\mathtt{b}_n)+\mathtt{rd}(\mathtt{b}_{n\text{-}\mathtt{down}}))
\end{aligned}$$

where $\mathtt{b}_{\mathtt{n-left}}$ is the left-neighbouring region of $\mathtt{b}_n$ and so forth, depending on the indexing scheme chosen. For a single cycle within a region, we apply $\mathtt{newvaluer}_n$ to all the points within the region:

$$\mathtt{calcr}_n = \mathtt{newvaluer}_n \| \mathtt{newvaluer}_n \| \ldots$$

For the four-processor decomposition, we would perform a single cycle of the computation by applying $\mathtt{calcr}_n$ to all four sets of sub-areas. The full two-cycle computation is given by:

$$\begin{aligned}
&\mathtt{waver} = \upsilon\mathtt{a}.\upsilon\mathtt{b}.\upsilon\mathtt{c}.(\mathtt{setup};\mathtt{twostepr})\\
&\mathtt{twostepr} = \mathtt{step1};\mathtt{step2}\\
&\mathtt{step1} = \Delta_{\mathtt{a}}\{\mathtt{a}_1,\mathtt{a}_2,\mathtt{a}_3,\mathtt{a}_4\}.\Delta_{\mathtt{b}}\{\mathtt{b}_1,\mathtt{b}_2,\mathtt{b}_3,\mathtt{b}_4\}.\Delta_{\mathtt{c}}\{\mathtt{c}_1,\mathtt{c}_2,\mathtt{c}_3,\mathtt{c}_4\}.\mathtt{calcr}\\
&\mathtt{step2} = \upsilon\mathtt{a}'.\upsilon\mathtt{b}'.\upsilon\mathtt{c}'.(\mathtt{cp}(\mathtt{b},\mathtt{a}');\mathtt{cp}(\mathtt{c},\mathtt{b}');\\
&\qquad\qquad \Delta_{\mathtt{a}'}\{\mathtt{a}_1,\mathtt{a}_2,\mathtt{a}_3,\mathtt{a}_4\}.\Delta_{\mathtt{b}'}\{\mathtt{b}_1,\mathtt{b}_2,\mathtt{b}_3,\mathtt{b}_4\}.\Delta_{\mathtt{c}'}\{\mathtt{c}_1,\mathtt{c}_2,\mathtt{c}_3,\mathtt{c}_4\}.\mathtt{calcr})\\
&\mathtt{calcr} = \mathtt{calcr}_1\|\mathtt{calcr}_2\|\mathtt{calcr}_3\|\mathtt{calcr}_4
\end{aligned}$$

We have not yet identified the halos explicitly. This is a useful thing to do, as it defines exactly which parts of a sub-grid are needed in computations on other sub-grids, which in turn allows optimisation of the sharing. The grid represented by $\mathtt{b}$ has been divided into four sub-grids $\mathtt{b}_1$–$\mathtt{b}_4$. Each of these sub-grids is further divided into three parts: vertical halo, horizontal halo, and non-halo elements. Note that these divisions are *not* disjoint, as the two halo areas share an element in the corner. We identify the two halo regions within an area $\mathtt{b}_n$ by $\mathtt{b}_{nv}$ and $\mathtt{b}_{nh}$ for the vertical and horizontal halo areas respectively. Using this decomposition$\mathtt{newvaluer}_n$ may be re-written as

$$\begin{aligned}
&\mathtt{newvaluer}_n' = (\mathtt{rd}(\mathtt{b}_n)\|\mathtt{rd}(\mathtt{a}_n)\|\mathtt{rdhalo}_n');\mathtt{wr}(\mathtt{c}_n)\\
&\mathtt{rdhalo}_n' = (\mathtt{rd}(\mathtt{b}_n)+\mathtt{rd}(\mathtt{b}_{\mathtt{nh-left}}))\|(\mathtt{rd}(\mathtt{b}_n)+\mathtt{rd}(\mathtt{b}_{\mathtt{nh-right}}))\|\\
&\qquad\qquad (\mathtt{rd}(\mathtt{b}_n)+\mathtt{rd}(\mathtt{b}_{\mathtt{nv-up}}))\|(\mathtt{rd}(\mathtt{b}_n)+\mathtt{rd}(\mathtt{b}_{\mathtt{nv-down}}))
\end{aligned}$$

where $\mathtt{b}_{\mathtt{nh-left}}$ denotes the horizontal halo of $\mathtt{b}_n$'s left neighbour and so forth. These expressions give rise to an expression $\mathtt{calcr}'$ for the full calculation. The expression $\mathtt{step1}$ may be re-written to use halos:

$$\begin{aligned}
&\mathtt{step1}' = \Delta_{\mathtt{a}}\{\mathtt{a}_1,\mathtt{a}_2,\mathtt{a}_3,\mathtt{a}_4\}.\Delta_{\mathtt{b}}\{\mathtt{b}_1,\mathtt{b}_2,\mathtt{b}_3,\mathtt{b}_4\}.\Delta_{\mathtt{c}}\{\mathtt{c}_1,\mathtt{c}_2,\mathtt{c}_3,\mathtt{c}_4\}.\\
&\qquad\qquad \Delta_{\mathtt{b}_1}\mathtt{b}_{1h}.\Delta_{\mathtt{b}_1}\mathtt{b}_{1v}.\ldots.\mathtt{calcr}'
\end{aligned}$$

We may show that $\mathtt{step1}'$ only ever accesses sub-areas of each $\mathtt{b}_n$, not the full areas. Furthermore, no process ever causes write events within any $\mathtt{b}_n$ in the scope of $\mathtt{step1}'$. We may therefore introduce copy actions to move the halo regions used by each sub-calculation into a local area. To make the derivation simpler to read, we shall abstract from $\mathtt{step1}'$ the terms which relate to the calculation of area$\mathtt{c}_1$, and transform them:

$$\begin{aligned}
\mathtt{step1}' &= \ldots\Delta_{\mathtt{b}_1}\mathtt{b}_{1h}.\Delta_{\mathtt{b}_1}\mathtt{b}_{1v}.(\mathtt{newvalue}_1\|\ldots)\\
&= \ldots\Delta_{\mathtt{b}_1}\mathtt{b}_{1h}.\Delta_{\mathtt{b}_1}\mathtt{b}_{1v}.(\upsilon\mathtt{h}.(\mathtt{cp}(\mathtt{b}_{2h},\mathtt{h});\alpha_{\{\mathtt{b2h}\mapsto\mathtt{h}\}}(\mathtt{newvalue}_1)))\\
&= \ldots\Delta_{\mathtt{b}_1}\mathtt{b}_{1h}.\Delta_{\mathtt{b}_1}\mathtt{b}_{1v}.\ (\upsilon\mathtt{h}.(\mathtt{cp}(\mathtt{b}_{2h},\mathtt{h});\upsilon\mathtt{v}.(\mathtt{cp}(\mathtt{b}_{3v},\mathtt{v});\\
&\qquad\qquad \alpha_{\{\mathtt{b3h}\mapsto\mathtt{h},\ \mathtt{b3v}\mapsto\mathtt{v}\}}(\mathtt{newvalue}_1\|\ldots))))\\
&= \ldots\Delta_{\mathtt{b}_1}\mathtt{b}_{1h}.\Delta_{\mathtt{b}_1}\mathtt{b}_{1v}.(\upsilon\mathtt{h}.(\mathtt{cp}(\mathtt{b}_{2h},\mathtt{h});\upsilon\mathtt{v}.(\mathtt{cp}(\mathtt{b}_{3v},\mathtt{v});\\
&\qquad\qquad \alpha_{\{\mathtt{b2h}\mapsto\mathtt{h},\ \mathtt{b3v}\mapsto\mathtt{v}\}}(\mathtt{newvalue}_1))))\|\ldots)
\end{aligned}$$

The newvalue$_n$ terms make use of the correct halos, copied into new areas used only by them. This means that the calculation of step1' may be re-written so that each region calculation first copies its halo into a local sharing area before using it, and does not access any other regions in the course of its computation. This models pre-fetch copying of data into local memory. Once more, the transformations are only effective because of a precise knowledge of the update behaviour of the underlying functionNewValue.

5 RELATED WORK

Research on process algebra has traditionally focused on calculi using communication between processes – indeed, our system is the only process algebra of which we are aware which addresses shared data. We see sharing theory as a possible complement to the usual algebras in the specification of shared memory computations.

Parallelising compilers make use of many of the optimisations we have identified. We believe that our theory allows many of the techniques of parallelisation, data dependence analysis (Zima, 1991) and interference analysis (Lucassen, 1988) to be cast in a new and more tractable framework. Furthermore, the theory gives insights into the design of types and operations which may eliminate much complex analysis by making the sharing behaviour of functions available directly to the compiler. In many ways this is closely related to algorithmic skeletons and bulk data types (Bird, 1986)(Skillicorn, 1991)(Skillicorn, 1995) with the important addition of being applicable to mutable data types.

Another related issue is that of weak memory coherence (Frank, 1992)(Li, 1989) caching, and bulk synchrony (McColl, 1994). We may use copy events to model the action of systems where "shared" state is not updated synchronously across a system, although the correspondence is far from exact and needs further investigation.

6 CONCLUSIONS AND FUTURE WORK

We have described the development of a theory of sharing in distributed systems, using a modified process algebra which allows shared pieces of state to be defined and manipulated. The core theory can describe programs performing read and write access to unitary pieces of shared data. Extensions allow shared data to be decomposed and atomic copies to be made. The theory can easily detect common synchronisation problems, and can be used to transform systems which use local caches of read-only data.

Our approach is to define high-level shared abstract data types whose definitions capture the most common programming idioms, including their sharing behaviour. We see sharing theory as applicable in three ways:

- in specifying the sharing behaviour of operations of types;
- in analysing new operations for unwanted interactions or potential bottlenecks, to ensure the type is scalable; and
- in defining new operations.

The first application uses the theory as a concise description of a function's interactions and side effects, and is close to the traditional uses of process algebra in specification. The second helps ensure that any types included in a distributed applications library are indeed scalable. The third –

rather more speculative – allows the programmer to define new operations and associate sharing expressions with them (or derive them automatically), making new functions "first class citizens".

Our immediate plans for the future include investigating performance models to guide analysis, for example to differentiate between possible and advantageous opportunities for caching. This will lead to methods to aid the design of types suitable for portable distributed programming, and the development of automated tool support for the analysis and manipulation of sharing expressions.

7 REFERENCES

Baeten, J.C.M. and Weijland, W.P. (1990) Process algebra. Cambridge University Press.

Bird, R. (1986) An introduction to the theory of lists, in *Logics for Programming and Calculi of Discrete Design.*

Frank, S. (1992) Virtual memory to ALLCACHE memory, in *Proceedings of the Virtual Shared Memory Symposium*, Centre for Novel Computing, University of Manchester

Li, K. and Hudak, P. (1989) Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, **7**, 243–271.

Lucassen, J.M. and Gifford, D.K. (1988) Polymorphic effect systems, in *Proceedings of the 15th ACM Symposium on Principles of Programming Languages.*

McColl, W.F. (1994) BSP programming, in *DIMACS series in Discrete Mathematics and Theoretical Computer Science.*

Milner, R. (1986) A calculus of communicating systems. Technical report ECS-LFCS-86-7, Laboratory for Foundations of Computer Science, University of Edinburgh.

Skillicorn, D.B. (1991) Models for practical parallel computation. *International Journal of Parallel Programming*, **20**, 133–158.

Skillicorn, D.B. (1995) Categorical data types, in *Abstract Machine Models for Highly Parallel Computing* (ed. J.R. Davy and P.M. Dew), Oxford Science Publishers.

Zima, H. and Chapman, B. (1991) Supercompilers for parallel and vector computers. ACM Press.

8 BIOGRAPHIES

Simon Dobson received a DPhil in Computer Science from the University of York in 1993, with a thesis on programming models for highly scalable computers. He joined the Rutherford Appleton Laboratory as a research fellow in 1992 to pursue his interests in languages and architectures for parallel and distributed systems, and has worked on a variety of projects involving advanced compilation techniques, system architectures, formal methods and hypermedia.

Chris Wadsworth started his research career in the Programming Research Group at Oxford University and worked at Syracuse and Edinburgh before moving to the Rutherford Appleton Laboratory in 1981 where he heads the Parallel and Distributed Systems Group. He is well known for his seminal contributions to lambda calculus, lazy evaluation and denotational semantics, and was a joint developer of the LCF theoremprover and the ML language.

16
Using Concurrency and Formal Methods for the Design of Safe Process Control

Thierry CATTEL
Laboratoire de Télinformatique, Ecole Polytechnique Fédérale
CH-1015 Lausanne, Switzerland
Tel. +41 21 693 67 76, Fax +41 21 693 66 00, E-mail cattel@di.epfl.ch

Abstract
This paper reports an experience with the modeling, verification and concurrent implementation of a medium-sized process control problem. The case study was proposed by Forschungszentrum Informatik, Karlsruhe in 1993 in order to promote the usage of formal methods in industry. It concerns an industrial robotics application that processes metal plates. A top-down design approach is followed where successive CCS and Promela specification levels of decreasing abstraction are considered, each layer little by little allows verification of parts of the security requirements thus providing a mean for coping with state explosion. The level refinements are checked with the Concurrency Workbench a CCS-based tool. Safety and liveness requirements are expressed in linear temporal logic and checked with SPIN. From the ultimate specification, two different implementations are derived. The first one is in Synchronous C++, a concurrent extension of C++ and the second in Regis/Darwin. This application shows that formal methods are quite appropriate for developing control process problem from scratch and with requirements to be checked in mind. It appeared clearly that the specification phase was very important for obtaining a satisfactory specification from which a well behaved implementation was derived easily in a few days.

Keywords
application, process control, refinement approach, linear temporal logic, process equivalence, concurrent programming.

1 INTRODUCTION

Though SPIN (Holzmann, 1991) was specially designed for tackling protocols, it appeared that it was also quite suitable for addressing other problems such as distributed algorithms and multiprocessor operating systems (Cattel, 1994). Provided one is able to express problems as protocols it is quite possible to take advantage of SPIN's power for modeling and verifying them also. We show in this paper that process control systems may be seen as particular protocols and be verified as such. Some dedicated languages and tools such as LUSTRE (Halbwachs, 1993) are certainly more powerful and efficient for expressing and verifying such systems, but it is not clear that the resulting specifications are more readable than the ones obtained with Promela. Furthermore there is no possibility, in particular with LUSTRE, to check liveness properties, whereas SPIN is well adapted for this purpose. When we started to design a controller for the Production Cell case study proposed in 1993 by Forschungszentrum Informatik, Karlsruhe, Germany (Lindner, 1993) there already existed around 20 contributions (Lewerentz, 1994). Few of them proposed a complete solution to the problem in term of specification, verification and implementation, and no one really solved the liveness requirements checking. My main motivation was to contribute by firstly addressing the liveness concerns. Another goal was to derive a straightforward implementation of the detailed Promela specifications by translation into Synchronous C++ (Call, 1994), a concurrent extension of C++ developed in our Labs, that should be soon integrated in Gnu distribution. We have also derived an implementation of the production cell in Regis/Darwin the distributed framework of Imperial College (Magee, 1995), from the Promela specifications; one of the advantages is that we obtained a clearer architecture that may be graphically built with a visual tool such as the Software Architect Assistant (Keng, 1995).

For being tractable the produced models need to be designed according to several levels of decreasing abstraction, thus a refinement approach was used. Unfortunately SPIN currently provides no way of verifying the consistency of such refinements since no support for checking process equivalencies is available. Some attempts exist (Erdogmus, 1995) for extending SPIN in that direction but only allows for restricted equivalencies (trace equivalence, trace inclusion). For this reason we also developed in parallel some CCS (Milner, 1989) models that correspond to the Promela specification.

In the following sections, we will first briefly present the production cell case study, the design approach used and some of the resulting models, the verification of the liveness requirements, the verification of some safety requirements and eventually some considerations related to the implementation. A brief introduction to Promela/SPIN is also provided.

2 THE PRODUCTION CELL CASE STUDY

This case study is inspired of an actual industrial installation in a metal-processing plant in Karlsruhe. It is a realistic industry-oriented problem in which safety requirements are important. The production cell (see Fig. 1) processes metal blanks with a press. First the blanks are introduced on a feedbelt that leads them towards a rotary table that presents the blanks to a two-armed robot. The robot takes the blanks from the table, feeds the press and takes back the forged blanks to deposit them on a second belt. This belt leads the blanks toward a travelling crane that brings them to the feedbelt again. The production cell is cyclical only for sake of the case study, in reality the blanks would be dropped from the travelling crane into some container. Up to 8 blanks may be processed in parallel by the system.

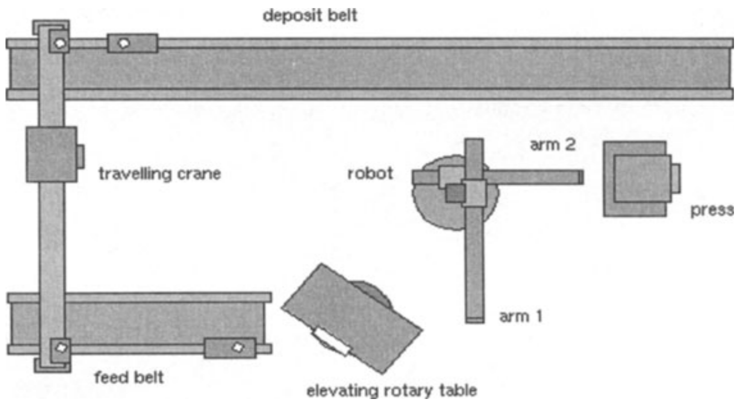


FIGURE 1 The Production Cell

The task description document (Lindner, 1993) describes three kinds of requirements: safety properties, liveness properties and performance and general software engineering properties. There are four classes of safety properties. First, mobility restriction properties that specify that a physical element has to restrict its movements within certain limits so as not to damage itself. Second, machine collision avoidance properties: for instance the press must not move if one arm of the robot is engaged inside it. Third, blanks must not be dropped outside safe regions, and fourth, to being distinguished, blanks need to be kept sufficiently distant from each other. The stronger liveness requirement expresses that every unforged blank introduced into the system will eventually leave it forged. A weaker form of this liveness property expresses that for each of the cell elements, if a given blank enters it, it will eventually exit it. The performance requirements are irrelevant here since we only address an untimed model. The software engineering considerations are related to the maintainability and flexibility of the resulting control software.

3 DESIGN APPROACH

First of all, we make the hypothesis that the controller we will design will be fast enough for controlling the plant, namely it will not loose any significant plant data and hence the corresponding models will be only qualitative and untimed. Second, we will comply with the following guidelines: as commonly accepted for reactive systems, the specification of the controller and the environment it drives will be clearly separated, besides the controller will be object-oriented, namely the architecture of the controller will reflect the physical cell organization in the sense that to each physical cell element will correspond an (active) object. And third, for the purpose of liveness checking, we will consider two versions of the system: a closed version where the travelling crane takes back the blanks to the feedbelt and an opened one where the travelling crane drops the blanks into a container.
The complete approach involves 4 description levels of decreasing abstraction. The motivation for doing so is to take advantage of a stepwise refinement development. Indeed this framework powerfully copes with complexity in two ways. First, the specifications are built progressively adding details to a very abstract model, thus relieving the designer of creative effort. The proof of the refinement correctness from one level to the next one consists in checking process equivalencies. It was carried out with the Concurrency Workbench (Cleaveland, 1993) on CCS specifications. Second, the successive description levels allow for checking included subsets of the requirements, thus giving a means for addressing the state explosion problem. More precisely the first description level only expresses abstract interactions (blank exchanges) through synchronous rendezvous between components of the controller, the second level integrates detailed interactions (adding for instance *"are you ready"* synchronization's before the effective blank passing), the third level is enriched with the movement orders submitted by the cells elementary controllers and the abstract specification of the movement realization. Finally the fourth level details the movement realization and includes the physical cell environment specification. It is from this last description, structured as three layers, of course without the environment specification, that a concurrent implementation will be derived in a straightforward way.

4 PROMELA/SPIN

SPIN (Holzmann, 1993, 1995) is a generally distributed automated verification system that is slowly evolving into an academic and industrial standard for on-the-fly LTL model checking. The two main advantages of the tool are that it is firmly founded on formal automata theory, and it can handle applications of full-scale industrial size. The tool is still evolving, adopting new advances in automata theory as they develop, and is therefore of growing interest to both theoreticians and practitioners.

The description formalism Promela (Holzmann, 1991) was designed as a subset of SDL. It allows express a concurrent or distributed system as a set of independent processes (*proctype*) communicating through synchronous (rendez-vous) and ansynchronous channels (*chan*). The data types available are restricted to simple types such as integers and boolean and may serve for building richer types with structure and array constructors. Promela's syntax is very close to C's and CSP's. There are also extra statements for defining *atomic* action sequences. It is possible to specify logical assertions (*assert*) inside the models and also more general linear temporal requirements applying to state sequences. Some labels are provided for marking the system states in order to track deadlocks (*end*) or cycles without progress (*accept*, *progress*). We present below (Fig. 2) a simple Promela specification of a four processes system. Three *user* processes compete in accessing a critical section in mutual exclusion. The fourth process *dijkstra* implements a semaphore as a synchronous communication channel on which *p* and *v* messages are sent (*!*) and received (*?*). The integer state variable *in_critical_section* records the number of users in critical section and the assertion specifies the mutual exclusion requirement.

```
mtype = {p,v};
chan semaphore = [0] of {mtype};
byte in_critical_section;

proctype dijkstra()
{ do
  :: semaphore!p -> semaphore?v
  od}

proctype user()
{ do
  :: semaphore?p;
     /* critical section */
       in_critical_section = in_critical_section + 1;
       assert(in_critical_section == 1);
       in_critical_section = in_critical_section - 1;
       semaphore!v
     /* end of critical section */
  od}

init{
  atomic{
    run dijkstra();
    run user(); run user(); run user()
  }
}
```

FIGURE 2 A Simple Mutual Exclusion System

The SPIN tool is composed of a simulator, a model-checker and a graphical debugger. From a Promela model is possible to do simulations according to several visual modes with a selective amount of details (communication interactions, state variables,...). More interesting is the model-checker that walks through systems of several millions states while checking for deadlocks, assertion or temporal claim violations. If an error is found, a trace is stored that guides the simulator for a diagnosis session. SPIN is provided with two optimization algorithms that insures its great efficiency: one for random-walk searches and another one for partial-order reduction techniques. The tool also provides the facility to express LTL (Manna, 1992) properties at a high level and automatically translates them into Büchi automata for model-checking purpose. As an example here is the general expression of a progress requirement. specifying that every *p*-state is followed by a *q*-state: *[](p -> <>q)*

5 DESIGN, MODELLING AND VERIFICATION

The Promela models were written for levels 2 to 4. With level 2 it was possible to check that the closed production cell may process up to 8 blanks without deadlocking (it reaches a deadlock as soon as the 9th blank arrives), and to check the strong liveness requirements with a 100% coverage with supertrace on a 128M memory machine. If the number of blanks is limited (e.g. to 4), full search (without supertrace) may be done. Attempts to check these properties on models of further levels is possible but the coverage decreases dramatically from the second or third blank.
From level 3 the weak liveness requirement and the machine collision avoidance requirements may be verified as well as the properties expressing that the blanks are dropped only in safe regions. With level 4 it is possible to check that the blanks are kept sufficiently distant and that the element mobility restriction is insured. For all these, the full search is possible. Additionally it is verified that the specification does not contain any unreached code.
We now present some limited excerpts of the Promela models that will be used for showing the verification of some significant requirements.
First, follow the abstract models of the two ends of the opened production cell: the feedbelt and a container which would be placed after the travelling crane. The feedbelt is a one direction moving belt with a motor that may be set to *on* or *off*. At its end a sensor detects the incoming blanks. When a blank arrives at the sensor the belt stops and continues if the next element, the rotary table is ready for accepting it. A blank may be put at the entrance of the feedbelt if it is empty or as soon as the previous blank has reached the sensor, so this belt may contain two blanks at the most. It is modelled with two processes (Fig.3).

```
proctype FeedBelt1(chan in1,out)
{ byte blanknum;
  do
#if !LIVENESS
     :: in1?blank(blanknum,recvblankforged);
        atomic{
          prevrecvblank=recvblank;
          recvblank=blanknum;
          assert(recvblank==(prevrecvblank+1)%MAX_BLANKS;
        };
        out!blank(recvblank,0);
#else
     :: in1?blank(recvblank,recvblankforged);
        out!blank(recvblank,0);
        recvblank=UNDEF;
#endif
  od;
}
proctype FeedBelt2(chan in,out)
{ byte blanknum;
  bit forg;
  do
  :: in?blank(blanknum,forg);
     out!ready(0,0);
     out!blank(blanknum,forg);
  od;
}
```

FIGURE 3 FeedBelt

```
proctype Container(chan in)
{ do
#if !LIVENESS
  :: in?ready(0,0);
     prevworkblank=workblank;
     in?blank(workblank,workblankforged);
     assert(workblank==(prevworkblank+1)%MAX_BLANKS);
#else
  :: in?ready(0,0);
     in?blank(workblank,workblankforged);
#endif
  od;
};
```

FIGURE 4 Container

The production of the blanks is modelled by a process that sends them to the feedbelt; they are materialized as two-fielded messages, the first field being a blank number modulo *MAX_BLANKS*, the second is a boolean telling if the blank is forged or not. The container accepts all the incoming blanks, it is modelled by process of Fig.4.
The verification of the strong liveness requirement, may be achieved thanks to two safety properties and a progress property expressed in LTL (Manna, 1992). The first safety property means that there must not be any blank duplication nor blank loss, the second one means that the sequence of number of the sent blanks need to be preserved when they are received by the container. These two properties may be easily captured with the two assertions appearing in Fig.3 and 4 where *LIVENESS* should be set to *0*. The progress requirement is verified with the following. (Π stands for logical conjunction, *AG* for the temporal *"always"* operator (box), and *AF* for *"eventually"* (diamond):

$$\bigcap_{i=1}^{m} \Box(P(m) \rightarrow \Diamond Q(m)) \tag{1}$$

or equivalently

$$\Box\left(\bigcap_{i=1}^{m} P(m) \rightarrow \Diamond Q(m)\right) \tag{2}$$

where

$$\begin{aligned} P(b) &= ((recvblank = b) \wedge \neg recvblankforged) \\ Q(b) &= ((workblank = b) \wedge \neg recvblankforged) \end{aligned} \tag{3}$$

It is enough that m equals 7 if *MAX_BLANKS* is equal to 8. Fig.5 shows the related Promela definitions, including the Büchi automata for the progress requirement.

```
#define P(b) ((recvblank==b) && !recvblankforged)
#define Q(b) ((workblank==b) && workblankforged)
/*![]((P(0)-><>Q(0))&&((P(1)-><>Q(1))&& ... && (P(7)-><>Q(7)))) */
```

```
never{
    do
    :: skip
    :: (P(0)&&!Q(0)) -> goto accept0
    ...
    :: (P(7)&&!Q(7)) -> goto accept7
    od;
  accept0:
    do
    :: !Q(0)
    od;
  ...
  accept7:
    do
    :: !Q(7)
    od;}
```

FIGURE 5 Strong liveness requirement

We now make explicit the safety requirement related to the absence of collision between the press and the robot. A scenario is built with the robot and the press detailed models, and two processes that very abstractly emulate the output of the table to the robot and the input of the deposit belt from the robot. The safety property is verified with the LTL property:

$$\Box(pressing \rightarrow (\neg arm1\_in\_press \wedge \neg arm2\_in\_press)) \quad (4)$$

Where *pressing*, *arml_in_press* and *arm2_in_press* are variables set respectively by the press and the robot in appropriate situations.

We conclude this section with the process in charge of the execution of the table horizontal movements. It will show how the safety requirements regarding machine restriction mobility are defined and how the hypothesis of the controller being fast enough is actually modelled. The table may move horizontally in 2 directions thanks to a motor (*A7*) that may be set to *on*, *rev* or *off*. Two positions are significant: *left* and *right* given by a potentiometer sensor (*S9*). The process *TableH* accepts commands *go_right*, *go_left* on channel *in* and reports their completion on channel *out*.

```
/* specification */
proctype TableH(chan in,out)
  { do  :: in?go_right;out!at_right;  :: in?go_left;out!at_left; od}

/* detailed model */
proctype TableH(chan in,out)
{ byte command, ack;
  do
  :: in?command;
     atomic{
       A7=(command==go_right -> ON : REV);
       /*......................Environment..................*/
       if
       :: (S9==TA_LEFT && A7==REV)   -> assert(FALSE);
       :: (S9==TA_RIGHT && A7==ON)   -> assert(FALSE);
       :: (S9==TA_LEFT && A7==ON)    -> S9=TA_RIGHT;
       :: (S9==TA_RIGHT && A7==REV) -> S9=TA_LEFT;
```

```
          fi;
          /*.........................................*/
          if
          :: (S9==TA_LEFT)  -> A7=STOP;ack=at_left;
          :: (S9==TA_RIGHT) -> A7=STOP;ack=at_right;
          fi;
        };
        out!ack;
      od
  }
```

FIGURE 6 Table Horizontal Movements

Fig.6 shows the trivial specification of *TableH* and its detailed model: when a command is received the motor is set in the appropriate direction, then the part of the environment related to *TableH* is given the control and evolves according to possible physical changes. For instance when the Table is at *left* and the motor is *on*, then the Table will reach the *right* position. Then *TableH* gets back the control and reports the command completion. The attempts to exceed the physical limits are naturally expressed in the environment evolution possibilities, for instance if the table is at *left* and the motor is *rev*. One may notice how the environment specification is separated from that of the controller's and we will see in the implementation section how its suppression will lead to the actual controller. The fact that the controller and the corresponding environment are merged in a single process may surprise at first. This is just a commodity for reducing the verification complexity because they could have been put in separated processes communicating through rendezvous for exchanging the motor orders and the sensors values (this is actually the way it is expressed in the CCS models).
In both cases what is important is that the environment is constrained to evolve only under the supervision of the controller. We will see in the implementation section that this corresponds to the synchronous option for driving the graphical simulation of the production cell, thus preserving the hypothesis that the controller is fast enough with regard to the plant.

6 IMPLEMENTATION

Below, we outline how the implementations were derived from the Promela models. The physical production cell in reduction (Fischer Technik) or the graphical simulation written by FZI in Tcl/Tk (Ousterhout, 1994) can be driven with the same protocol. On UNIX, commands are sent via *stdout*, and sensor values are read from *stdin* after having sent the command *"get_status"*. In this implementation a particular process called *Sampler* regularly samples the environment and forwards the interesting values of the sensors to the concerned controllers (e.g. *S9right*, *S9left*). The use of the special command *"react"* and the option *"-snc"* for running the simulation in synchronous mode, insures that whatever the speed of the controller it will not lose any sensor values.
Synchronous C++, is an extension of C++ that is also very similar to Ada to given extents. Fig.7 shows the implementation of *TableH*.

```
active class TableH{
   Environment  *env;
   Table        *ta;
   command      com;
   @TableH(){
     position ack;
     ...
     for(;;){
       accept In;
```

```
        switch(com){
          case go_right:
            env->Putcommand("table_right");
             break;
          case go_left:
            env->Putcommand("table_left");
             break;
        };
        select{
          accept S9right;
          env->Putcommand("table_stop_h");
          tack=at_right;
         ||
          accept S9left;
          env->Putcommand("table_stop_h");
          tack=at_left;
        };
        ta->Hout(ack);
      };
   };
public:
   void In(command com) {this->com=com};
   void S9right() {};
   void S9left() {}; ...
};
```

FIGURE 7 TableH implementation

A process template is declared as an active class and instances are created with *new* as instances of usual C++ classes. Synchronous rendezvous are declared as methods of an active class. They correspond to Ada task entries and may be used within a *select* statement for awaiting multiple events with the *accept* clause. This may be done on a local rendezvous as on a call to a rendezvous of an other object, in that sense is Synchronous C++ more symetrical than Ada.
In Synchronous C++ it is possible to structure the program as a network of nested active objects but there is no dedicated support to set up connections between active objects; the consequence is that the programs tend to have a flat structure and one has to add extra statements for initializing the objects so that they know of each other if needed. This is sometimes tedious and leads to mixing up the program architecture with its behaviour.
Darwin (Magee, 1995) allows for expressing the architecture of a system as an interconnected network of components each of which may be itself a network of components. The interface of each component is a collection of input/output ports. Only for the leaves of such a system does one need to express the behaviour. This is done with Regis which is C++ with some predefined classes for managing the communication ports. Regis globally allows the same features as Synchronous C++, in particular it possesses an equivalent to the *select* statement but in a more hand-coded way than in Synchronous C++, besides it is possible to await for multiple events in reception but not in emission, whereas both are allowed in Synchronous C++. The implementation of *TableH* in Regis is similar to the one in Synchronous C++. Fig. 8 shows the overall architecture of the production cell application in graphical Darwin created with the Software Architect tool.

Fig.9 shows the corresponding textual Darwin description and also the way *TableH* is interconnected to the whole controller as well as its interface.

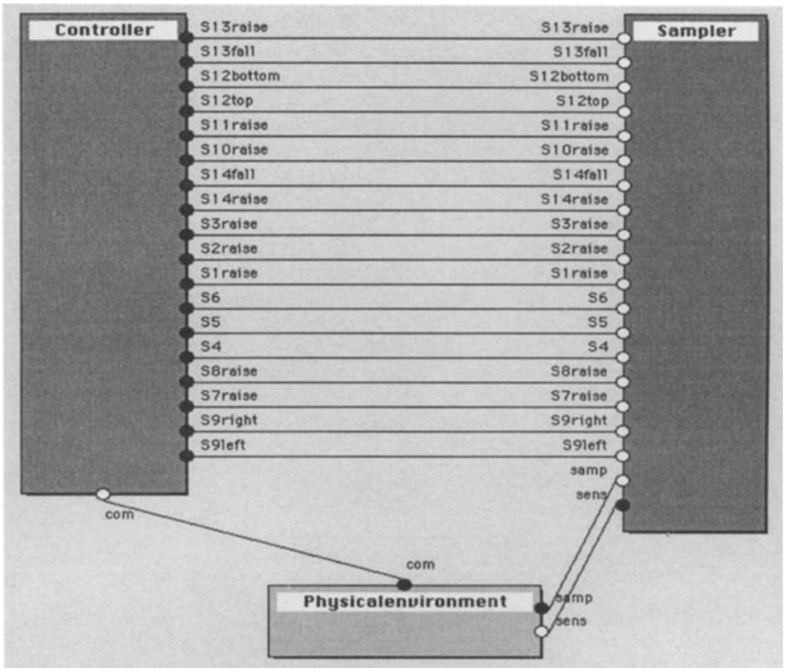


FIGURE 8 Graphical Darwin architecture of the Controller

```
/* Prodcell.dw */
#include "Controller.dw"
#include "Sampler.dw"
#include "Physicalenvironment.dw"
component ProdCell (int blanks=8)  {
  inst
    co : Controller(blanks);
    sa : Sampler;
    en : Physicalenvironment;
  bind
    sa.S13raise   -- co.S13raise;
    sa.S13fall    -- co.S13fall;
    sa.S9left     -- co.S9left;
    sa.S9right    -- co.S9right;
    sa.S7raise    -- co.S7raise;
    sa.S8raise    -- co.S8raise;
    sa.S4         -- co.S4;
    sa.S5         -- co.S5;
    sa.S6         -- co.S6;
    ...

    co.com       -- en.com;
    sa.samp     -- en.samp;
    en.sens     -- sa.sens;
}

/* Controller.dw */
#include "Table.dw"
#include "TableH.dw"
...
component Controller (int blanks)  {
  require
    com <port action>;
  provide
    S9left <port int>;
    S9right  <port int>; ...
  inst
    ta : Table;
    tah : TableH; ...
  bind
    S9left    -- tah.S9left;
    S9right   -- tah.S9right;
    ta.Hin   -- tah.in;
    tah.out  -- ta.Hout;
    tah.com  -- com; ...
}

/* TableH.dw */
component TableH  {
  provide
    in <port command>;
    S9left <port int>;
    S9right  <port int>;
  require
    out  <port position>;
    com <port action>;
}
```

FIGURE 9 TableH Implementation Architecture in Darwin

7 CONCLUSIONS

This work shows that there are great benefit in using formal methods and concurrency for process control with some property to be verified in mind, and that it fits well within a top-down design approach. There are lots of similarities between the verification of this case study and that of a protocol. First the problem was structured in layers of decreasing abstraction and second the strong liveness requirement verification was inspired from that of a sliding window protocol studied previously. Since the cell is considered in its normal functioning mode (no element breakdown or transmission failure), the problem was further simplified and no fairness consideration had to be taken into account.
The whole specification was written with only synchronous communications and its translation into a concurrent programming language was quite easy, the distance between both being very small. As forecast, no unforeseen events occurred when running the implementation since many design inconsistencies such as deadlocks, safety violations or cycles had been discarded during the specification phase. The specification and implementation are quite readable and the whole approach should be accessible to any engineer. The obtained models are easily modifiable and

generic enough to be reused in similar problems. Regarding the requirements, they were almost all specified in LTL. SPIN being now provided with a LTL translator, it is possible to remain at this high specification level. The new debugging facilities of Xspin (message charts, hypertext correspondence, breakpoints,...) where of great aid and we hope that SPIN will keep on growing, in particular toward verification of process equivalencies.
Some more details about this case study (reports, models, demos and code) may be found by FTP at *ltidec1.epfl.ch /pub* or by WWW at *http://ltiwww.epfl.ch/~cattel/prodcell.html.*

8 REFERENCES

Holzmann G.J. (1995) What's new in SPIN version 2, AT&T Bell Laboratories.

Holzmann, G.J. (1991) Design and Validation of Computer Protocols, Prentice Hall.

Holzmann G.J. (1993) Design and validation of protocols : a tutorial, in *Computer Networks*, 25(9), pp. 981-1017.

Cattel, T. (1994) Modelling and verification of a multiprocessor realtime OS kernel, in *Proceedings of the Seventh International Conference on Formal Description Techniques*, Berne, Switzerland.

Halbwachs N. (1993) Synchronous Programming of Reactive Systems, Kluwer Academic Publishers.

Lindner T. (1993) Production Cell Case study, Task description, ZFI, Karlsruhe.

Lewerentz C. and Lindner T. (1994) Formal Development of Reactive Systems, Case Study Production Cell, in *Lecture Notes in Computer Sciences 891*, Spinger Verlag.

Caal G., Divin A. and Petitpierre C. (1994) Active Objects: a Paradygm forCommunications and Event Driven Systems, in *Proceedings of Globecom'94*, San Francisco.

Magee J., Dulay, N., Eisenbach S and Kramer J. (1995) Specifying Distributed Software Architectures, in *Proc. of 5th Software Engineering Conference*, ESEC'95, Barcelona.

Keng N., Kramer J., Magee J. and Dulay N. (1995) The Software Architect's Assistant - A visual Environment for Distributed Programming, in *Proceedings of Hawaii International Conf. on System Sciences.*

Erdogmus H. (1995) Verifying Semantic Relations in SPIN, in Proceedings of 1st SPIN Workshop, Montreal.

Milner R. (1989) Communication and Concurrency, Prentice Hall.

Cleaveland R., Parrow J. and Steffen B. (1993) TheConcurrency Workbench : A semantics-Based Tool for the Verification of Concurrent Systems, in *ACM TOPLAS, Vol. 5, No 1.*, January, pp. 36-72.

Manna Z. and Pnueli A. (1992) The Temporal Logic of Reactive and Concurrent Systems - Specification. Springer-Verlag.

Ousterhout J. (1994) Tcl and the Tk toolkit, Addison-Wesley.

9 BIOGRAPHY

Thierry Cattel has obtained a PhD in Computer Science at the University of Franche-Comté, Besançon, France in 1992. He then has spent 18 months as an attached researcher at the Software Engineering Laboratory of the National Research Council of Canada, Ottawa where he aimed at specifying and verifying the Harmony real-time multiprocessor operating system. He has been first assistant at the Laboratoire de Téléinformatique, EPFL, Lausanne since September 1994. His research interests are Concurrent Software Engineering, Temporal Logics and Compositionnal Verification.

17
Using dataflow algebra to analyse the alternating bit protocol

A. J. Cowling & M. C. Nike
Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield, S1 4DP, U.K.
Telephone: +44 114 282 5580; Fax: +44 114 278 0972;
Email: A.Cowling @ dcs.shef.uk.ac

Abstract

The alternating bit protocol is taken as a case study of a parallel distributed system, and it is shown how the dataflow algebra approach can be used to specify and then analyse the overall behaviour of a communications system that uses this protocol. The paper summarises the use of dataflow algebras for specifying such systems, and introduces the main features of the protocol that are relevant to the case study. Models are developed for two different cases of the behaviour of the system, distinguished by different conditions on the length of the timeout period that is integral to the operation of the protocol. It is shown that under one of these conditions the overall operation of the protocol is such that it can not be guaranteed to operate correctly, even though the individual processes may operate correctly. A brief comparison is made between the use of the dataflow algebra approach for carrying out such analyses and the use of process algebra models.

Keywords

Dataflow algebra, process algebra, formal specification, formal verification, alternating bit protocol.

1 INTRODUCTION

The fundamental characteristic of parallel or distributed systems is that they consist of a number of concurrent processes which interact in some fashion, so that their correct operation depends not only on the correctness of the individual processes, but also on these processes interacting correctly. In principle it ought to be possible to analyse these interactions in either a top-down or a bottom-up fashion, but in practice the analysis methods available are nearly all bottom-up: that is, they start from descriptions of the behaviour of the individual processes, and then analyse the way in which these are composed to produce the behaviour of the complete system. These descriptions of the individual processes have to contain some representations of their abstract state spaces, and as the descriptions are combined the size of the composite state space for the whole system explodes combinatorially with the number of processes.

Consequently, for these methods the complexity of the analysis process grows similarly, unless ways can be found for pruning it, and so the starting point for this paper is the hypothesis that some form of top-down description of the allowable behaviour of the overall system is needed to guide this pruning. To justify this, the paper presents a case study of an approach that we are developing, based on what we call the dataflow algebra model. This is intended to be complementary to process algebra models, but it emphasises system-wide patterns of communication rather than those for individual processes. The example used in the study is the alternating bit communications protocol, which we have chosen largely because process algebra models of different versions of it have already received a lot of attention, as in Milner (1983), although without producing the results concerning the overall correctness (or lack of it) for the version of the protocol that we present here.

In section 2 of the paper, therefore, we firstly describe the dataflow algebra approach, and in particular explain how it can be used to provide two different levels of detail in the specification of a system, which we term the syntactic and the semantic levels respectively. Section 3 then summarises the important features of the alternating bit protocol, indicating the different variants of it that can exist and identifying the particular variant that is used for this case study. Sections 4 and 5 present specifications of the protocol at these two levels of detail, and then sections 6 and 7 show how these are analysed for two cases, the first representing a situation where the protocol can be guaranteed to operate correctly and the second case one where incorrect operation is proved to be possible. Finally in section 8 we make a comparison between this approach and ones based on process algebra, and summarise the conclusions to be drawn from this comparison.

2 DATAFLOW ALGEBRA MODELS

The basic concepts of dataflow algebra models are described in more detail in Cowling (1995), and are derived from the data flow diagrams used in nearly all systems analysis methodologies, such as SSADM (CCTA, 1990) or Yourdon (1989) and its derivatives. The fundamental model underlying the algebra is that a system consists of processes which communicate via unidirectional channels, so that the basic element of the algebra is an action which consists of a single message moving from a source process s to a destination process d via some channel c: this action is denoted s ! c ? d. At the syntactic level we are then concerned with describing the behaviour of a system in terms of the allowable sequences of such actions, which in CSP terminology (Hoare, 1985) would be the allowable traces, so that the dataflow algebra is essentially an algebra of traces. The basic operations of this algebra are thus the concatenation of two sequences of actions (written s1 ; s2) and the choice between two sequences (written s1 | s2): for the purposes of specifying a system this choice is treated as nondeterministic.

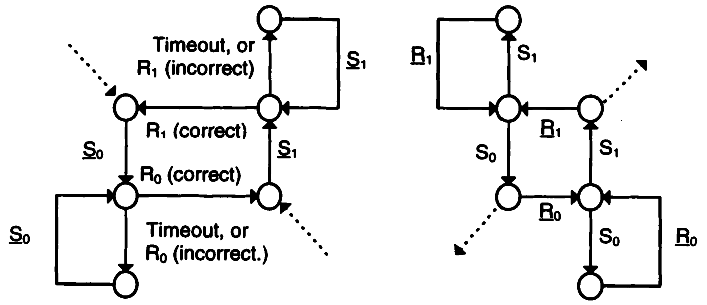
From these basic operations, and the silent action ε, two other main operations can be derived. One of these is repetition, so that s^n for any natural number n denotes n repetitions of s; also s^* denotes zero or more repetitions of s and s^+ denotes one or more repetitions of s. The other operation is parallel composition, written s1 || s2, which denotes the choice between any of the set of sequences obtained by an arbitrary interleaving of the actions of s1 and s2. As will be seen later, in some cases we need to restrict the set of possible interleavings, but the issue of how best to model these still requires further investigation.

A syntactic level specification of a system then consists of the set of expressions that define its permitted traces, and this can also be thought of as forming the production rules for a grammar which will generate the allowable set of traces. Then, what we call a semantic level specification (which is not the same thing as the sets of traces, even though these would often be understood as constituting the semantics of the grammar) can be developed by incorporating into this description specifications of the data that is contained in the messages. These specifications will define both the types of the data that can be communicated along each channel, and the processing which must be carried out in order to produce the data items that are output onto a channel and to handle the data items received from a channel. Components of these are embedded in the basic grammar to form an attribute grammar, in the same way as attribute grammars for programming languages are used to embed semantic definitions into their syntactic descriptions.

One effect of this embedding is that the sequences of actions define orderings on the execution of the various functions that specify the processing, and these orderings can be used in constructing functions to represent the overall processing of the system. Defining and using such orderings seems to us to be one of the fundamental problems in reasoning about the behaviour of parallel systems, and the approach that we are adopting here is intended to be applicable to any formal specification methodology that could be used for defining the individual components of the processing. In the example presented here, the attributes will be used to embed specifications that have been written in OBJ (Goguen and Winkler, 1988).

3 THE ALTERNATING BIT PROTOCOL

The alternating bit protocol (ABP) was introduced by Bartlett, Scantlebury and Wilkinson (1969) as a means of transmitting data reliably across an unreliable medium. It is an automatic repeat request protocol, in which both the packets of data and the replies carry a single-bit sequence number (the alternation bit) which is used to encode whether or not data has been correctly received on the other side of the medium. In the original description it is assumed that the data contains check bits from which the receiver determines whether to request a retransmission, but in fact that original description would be equally valid if the receiver is merely expected to echo the data back, so that the transmitter checks whether or not it matches what was originally sent.



(a) The Sender Process (b) The Receiver Process

Figure 1 Finite State Automata for the Alternating Bit Protocol.

In Figure 1 the finite state automata for the sender (a) and receiver (b) processes are shown. These are similar to the diagrams in the original, except that here data is only being passed in one direction, and the timeout has been included on the relevant arcs. The labels on the arcs represent values being received or sent by each process, with S denoting data packets and R denoting replies.

The underlined labels represent transmissions, and the subscripts represent the values of the alternation bit for the messages. The dotted arrows in (a) denote the acceptance of a new packet for transmission, after a reply has been received that indicates that the packet had been received correctly. Delivery of the item currently being sent occurs when the receiver accepts a packet with the alternation bit "flipped", and the dotted arrows in (b) show where this happens.

4 A SYNTACTIC SPECIFICATION

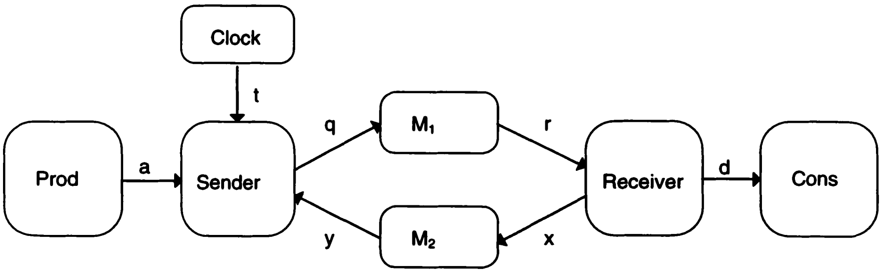


Figure 2 Data Flow diagram for the Alternating Bit Protocol system.

Figure 2 shows the data flow diagram for a system using this protocol, with a producer process Prod using the intermediate processes to transmit packets to a consumer process Cons. For simplicity, it is assumed that the medium can be modelled as two separate processes, M_1 and M_2, representing respectively the medium for the journey from the sender to receiver and vice-versa. The medium can lose or corrupt packets, but it is assumed that it can not corrupt the value of the alternation bit in a packet. Also for simplicity, only a single channel is shown from the Clock, on which timeouts are signalled to the Sender: in practice there would need to be a reverse channel as well, on which messages were sent to start the clock timing. The topology of this system can then be defined by listing the basic actions that can occur, as follows: these will be the terminal symbols for any syntactic specification of the system.

ToS = Prod ! a ? Sender	timeout = Clock ! t ? Sender
S_1 = Sender ! q ? M_1	S_2 = M_1 ! r ? Receiver
R_1 = Receiver ! x ? M_2	R_2 = M_2 ! y ? Sender
Deliver = Receiver ! d ? Cons	

4.1 Simple Cyclic Operation

If no corruption or loss occurs, then the operation of the protocol will be cyclic, so that in terms of these actions the basic loop for sending one packet would be denoted S_1 ; S_2 ; R_1 ; R_2. This simple operation does, however, depend on the value of the timeout being greater than the maximum time that can be taken for the whole loop, so that no parallel actions can take place. Under this assumption, the full protocol ABP can be described as follows:

Start = S_1 ; (timeout ; S1)* ; S_2
Reply = R_1 ; (timeout | R_2)
OneCyle = MStart ; Reply ; Loop1*

MStart = ToS ; Start ; Deliver
Loop1 = Start ; Reply
ABP = OneCyle^n

The basic operation of this is that the ToS action represents a new value being accepted for transmission, and the flipping of the alternation bit. Any packet loss results in a timeout, as in both Start (where the subsequent restarting of the cycle is shown explicitly) and Reply (where the restarting is implicit in the definition of OneCycle). When a packet with this new bit is received across the medium, the Receiver delivers the currently held message and holds the new one until the bit flips again. The Loop1 component describes the fact that after the Deliver there may be multiple retransmissions of the same piece of data: once they finish OneCycle is over and a new one can begin. Therefore, ABP is simply an arbitrary number of executions of this cycle.

4.2 Parallel Operation

The time taken for the basic cycle (ie S_1 ; S_2 ; R_1 ; R_2) is not usually deterministic, and so if the timeout period were to be gradually reduced there would come a point in the operation where a timeout would sometimes occur part way through a cycle that had not yet finished, and so would trigger the start of a new cycle in parallel with the existing one. The significance of this case is that it could give rise to unreliable operation, as a consequence of the system-wide behaviour being incorrect rather than because of any failure of an individual process. To see how this could occur requires a more elaborate model of the overall behaviour, to describe the possible parallelism: a time sequence diagram for the actions involved is shown in Figure 3.

If we assume that the timeout period is larger than the time for a "half loop", but smaller than the time for a whole loop, then essentially the parallel operation can start once an outward message has reached Receiver, as represented in the diagram by the first part of the main sequence, labelled Start 1. At that point a timeout could occur, indicated by the dotted line, and this would cause a duplicate sequence to start up, labelled Duplicate 1: this operates in parallel with the reply being transmitted back as part of the main sequence (labelled Rest 1). Any further timeout, however, could not occur as part of this duplicate sequence, as by then the original cycle would have finished (indicated by the dotted line after Rest 1) and a new main loop would have started, labelled Start 2: consequently any such timeout would form part of that, indicated by the dotted line after Start 2. Under these assumptions, therefore, there can at most be two activities occurring in parallel. (Further shortening the timeout period could give rise to the possibility of more parallel activities, but we shall not consider that case in this paper.)

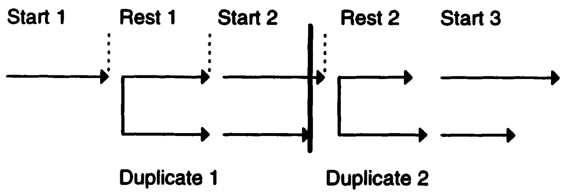


Figure 3 Time sequence diagram for parallel operation.

Thus, while the parallel activity of the duplicate sequence can continue beyond the start of the next main loop, it can only overlap with it so far, indicated by the thick vertical line. In principle, therefore, it must have finished before the point in the main loop at which another duplicate could start (ie the start of Rest 2). In practice, even though the sequence must have finished by then, its effects may not have, in that the duplicate may have returned a message to the Sender which effectively fools it into acting as though the new main sequence has already finished, and so causes it to send another message out. This is, however, modelled satisfactorily by the possibility of a new duplicate sequence starting up, as indicated by the start of Duplicate 2, except that in this case there will be no timeout involved, either in the duplicate sequence or the main one (ie in this case the dotted line before Rest 2 would not correspond to a timeout). Thus, although some duplicate sequences will start with a timeout, not all of them will. The duplicate sequence can therefore be defined as follows.

Duplicate = (timeout | ε) ; S_1 ; DupRest   DupRest = ε | S_2 ; R_1 | S_2 ; R_1 ; R_2

For the main loop there are four different cases that need to be considered, although not all are shown in detail in Figure 3. The cases depend on whether or not the alternation bit has flipped at the start of the loop, and whether or not the medium M_1 loses the message on the first time round the loop. With Start and Reply defined as in the simple cyclic case above, these four cases can in principle be expressed as

Loop2 = ((Reply ; S_1 ; S_2) || Duplicate) -- bit not flipped, no loss
| (((Reply ; S_1 ; timeout) || Duplicate) ; Start) -- bit not flipped, loss
| ((Reply ; ToS ; S_1 ; S_2 ; Deliver) || Duplicate) -- bit flipped, no loss
| (((Reply ; ToS ; S_1 ; timeout) || Duplicate) ; Start ; Deliver) -- bit flipped, loss
ABP = MStart ; Loop2^n

In practice, though, there are two problems with this specification, both arising from the fact that the parallel composition operation s1 || s2 is defined in terms of arbitrary interleavings of the sequences s1 and s2. Here, since both Reply and Duplicate can contain the actions R_1 and R_2, an arbitrary interleaving of them could contain either R_1 ; R_1 ; R_2 ; R_2 or R_1 ; R_2 ; R_1 ; R_2. One problem is therefore that the first of these would only be valid if the medium M_2 had some internal buffering capacity, and while this might be the case in a complex communications system, it would not be true of the simplest form of medium. The other problem is that the second interleaving is valid, but only makes sense if the first R_1 were associated with Reply rather than Duplicate. We are still exploring various ways of expressing the sort of constraints that are required in these two cases, but space does not permit discussion of them here. We are, however, satisfied that they can be expressed, and that for this example it is legitimate to assume where necessary that the messages pass through the medium M_2 in the order that one would expect.

5 A SEMANTIC SPECIFICATION

In the grammar, an action describes the output of a value from one process and the input of this value into another process. The semantics of an action can therefore be described by defining functions which model the input/output behaviours of the component processes. If each process has an output and input function, then the behaviour of an action will consist of the input function of the second process applied to the result produced by the output function of the first process. Then, from these expressions describing the input and output of each action, expressions can be built up to describe the behaviour of sequences of actions, and so finally an expression can be built up describing the behaviour of the whole system. In this particular example, we want to use this expression to show that the stream of values input into the system by Prod are all output at the other end to Cons, without any corruption. This can be done in the simple cyclic case, but not in the case of parallel operation, where we shall show that the protocol can be unstable.

Attribute grammars (Knuth, 1968) can be used to describe this level of operation of the system, in similar fashion to their use for specifying compilation of programming languages and other computations that involve tree structures (Kastens, 1991). Thus, each basic action is required to have attributes defined for it that we shall call Src and Dest, whose values will be the appropriate functions. In addition, there will have to be global variables representing the effects of these on the states of the different processes, and these will need to be passed around as an attribute of the actions, and used to construct the corresponding attributes for complete sequences. The semantic specification will therefore be built up by constructing specifications for the individual processes, and then attaching the appropriate functions as attributes to the actions. Attributes for the sequences can then be synthesised from these.

5.1 Process Specifications

For each process, the specification is defined in terms of some suitable abstraction of its state, and so could in principle be treated as a hidden state algebra (Goguen, 1991), although in presenting this example it is simpler to have the state models explicit. For each specification, therefore, the state of the process will be denoted simply by State, and then input operations defined to correspond to the generators for this abstract state, and output operations defined as observers of the state. The equations of the specification will then define in abstract terms how the outputs are computed from the inputs, and so can be used to reduce expressions involving these outputs.

In what follows, the basic type Message is assumed to be defined as Bit × Value, with observer operations bit and value, and with an operation flip defined on the type Bit. In principle Value could be any suitable type, which in practice would probably be a string of characters: it is assumed here to include a null element ε which is different to any meaningful value that can be transmitted. Models are given for each of the four processes in the system.

The Media

These two processes are identical, and each requires an input function, which accepts a message and stores it in some internal "register" (ie its state), and an output function which delivers the contents of that register. Each also requires a "corruption" function, which randomly corrupts a message's value, but not its alternation bit, and which does so in such a way that corruption in one medium can not reverse the effects of corruption in the other. The signatures of and equations for these functions are as follows.

$M_1.\text{in} : \text{Message} \times \text{State} \rightarrow \text{State}$
$M_1.\text{out} : \text{State} \rightarrow \text{Message}$
$c : \text{Message} \rightarrow \text{Message}$

$M_1.\text{out}(M_1.\text{in}(a,b)) = c(a)$
$\text{bit}(c(x)) = \text{bit}(x)$

$M_2.\text{in} : \text{Message} \times \text{State} \rightarrow \text{State}$
$M_2.\text{out} : \text{State} \rightarrow \text{message}$
$d : \text{Message} \rightarrow \text{Message}$

$M_2.\text{out}(M_2.\text{in}(a,b)) = d(a)$
$\text{bit}(d(x)) = \text{bit}(x)$

value(c(x)) = value(x), or value(d(x)) = value(x), or
value(c(x)) $\neq$ value(x), randomly value(d(x)) $\neq$ value(x), randomly
value(c(x)) $\neq$ value(x) $\Rightarrow$ value(d(c(x))) $\neq$ value(x)

The Sender

The state of this process consists of a pair of registers: the first one to hold the data that is currently being sent across the network, and the second one for the currently returned value. The process then requires two input functions and one output function, with the following signatures:

Sender.in$_1$: Value $\times$ State $\rightarrow$ State	-- input of a new item of data from Prod
Sender.in$_2$: Message $\times$ State $\rightarrow$ State	-- input of an item of data from R$_2$
Sender.out : State $\rightarrow$ Message	

Note that there does not need to be an input function to correspond to the arrival of a timeout, because this is a control flow rather than a data flow, and so its arrival does not alter the data state of the process. In terms of the state these three functions can be defined as follows, and equations derived from the definitions for reducing expressions involving these functions.

Sender.in$_1$(x, (v$_1$,v$_2$)) = ((flip(bit(v$_1$)),x) , (bit(v$_2$), ε))
Sender.in$_2$(y, (v$_1$,v$_2$)) = **if** bit(y) = bit(v$_1$) **then** (v$_1$,y) **else** (v$_1$, v$_2$) **endif**
Sender.out((v$_1$,v$_2$)) = v$_1$

value(Sender.out(Sender.in$_1$(x, (v$_1$,v$_2$)))) = x (1)
bit(Sender.out(Sender.in$_1$(x, (v$_1$,v$_2$)))) = flip(bit(v$_1$)) (2)
Sender.out(Sender.in$_2$(x, (v$_1$,v$_2$))) = v$_1$ (3)

Here, rules 1 and 2 represent the activity of a newly accepted value x being ouput as a message, while rule 3 represents the retransmission of an already held value (ie the one in the first register) after a comparison has proved incorrect, or after a "rogue" reply has been discarded.

The Receiver

This process also needs a pair of registers: the first one to hold the data that is currently being received, and the second one forming an "output buffer" into which it moves the current message that it is holding (ready for delivery) when it receives a message with the bit flipped. A full analysis of whether the system could be made to operate correctly would require complete histories to be stored, but that is not necessary here.

In terms of the state changes, there need to be two basic input functions, one (in$_1$) for the case where the bit has not changed and the other (in$_2$) for the case where it has: the function in can then be synthesised from these. There will also be two output functions, one corresponding to a Deliver action and the other to the transmission of a reply. The signatures of these operations will then be:

Receiver.in$_1$, Receiver.in$_2$, Receiver.in : Message $\times$ State $\rightarrow$ State

Receiver.out$_1$: State $\rightarrow$ Value	-- output for "deliver" port
Receiver.out$_2$: State $\rightarrow$ Message	-- output for "retransmit" port

The equations defining these, and the reductions that can be made, are:

```
$Receiver.in_1(x, (v_1,v_2)) = (x,v_2)$              -- new message replaces old in "current" buffer
$Receiver.in_2(x, (v_1,v_2)) = (x,v_1)$              -- also old message ($v_1$) placed in "output" buffer
$Receiver.in(x, (v_1,v_2)) =$ if $bit(x) = bit(v_1)$
    then $Receiver.in_1(x, (v_1,v_2))$ else $Receiver.in_2(x, (v_1,v_2))$ endif
$Receiver.out_1((v_1,v_2)) = value(v_2)$             -- deliver "output" buffer value
$Receiver.out_2((v_1,v_2)) = v_1$                    -- current message output for retransmission
```

$$Receiver.out_2(\ Receiver.in(x_i, (v_1,v_2))\ ) = x_i \tag{4}$$
$$Receiver.out_1(\ Receiver.in_2(x_i, (x_{i-1},v_2))\ ) = value(x_{i-1}) \tag{5}$$

Here, rule 4 represents the retransmission of a received message back to the sender, where the same message is sent back irrespective of which of the two input functions was invoked. Rule 5 represents the delivery of the value of the old message held (viz x_{i-1}) in the case where a value has just been accepted with a different bit.

5.2 Actions And Their Attributes

A global attribute States contains the array of process states, where the elements will be denoted as though indexed by the process names. Components of the state of a process p will be denoted as States[p,1] and States[p,2], as we have not formally defined projection functions for the pairs. Then, for each action the attributes Src and Dest can be defined in terms of the operations introduced above, and hence its effect can be synthesised. Most of the actions will have the same basic form, and this is illustrated for R_2, which uses M_2.out and Sender.in_2:

```
**Action** $R_2$ **is** $R_2.Src = M_2.out(States[M_2])$
    $R_2.Dest = Sender.in_2(R_2.Src, States[Sender])$
    $States[Sender]=R_2.Dest$
**EndAction**
```

Similarly, the S_1 action uses Sender.out and M_1.in; the action for S_2 uses M_1.out and Receiver.in; and the action for R_1 uses Receiver.out_2 and M_2.in. The only exceptions to this form are ToS and Deliver, since we have not specified any abstract operations for Prod or Cons, and so have to define directly the values accepted or delivered. These actions will therefore be written as:

```
**Action** ToS **is** $ToS.Src = x_i$
    $ToS.Dest = Sender.in_1.(ToS.Src, States[Sender])$
    $States[Sender] = ToS.Dest$
**EndAction**

**Action** Deliver **is** $Deliver.Src = Receiver.out_1(States[Receiver])$ **EndAction**
```

6 ANALYSIS OF RELIABLE OPERATION

To show that a value will be delivered correctly, a sequence has to be "translated" into the expressions it forms, which can be illustrated with the following example:

ToS ; S_1 ; timeout ; S_1 ; S_2 ; Deliver ; R_1 ; R_2 ; ToS ; S_1 ; S_2 ; Deliver

Each ToS action marks the start of the loop OneCycle, and associated with it will be an invariant that for States[Sender,1], States[Sender,2] and States[Receiver,1] the bits must all be the same. Proving maintenance of this invariant will be an important part of proving correct operation for the overall system. We assume that Sender is initially in state (v,v), where v = $(0,x_{i-1})$, meaning that a succesful loop has been completed; and that Receiver is in the state (v,y), where bit(y) = 0. Using the attributes we can determine the state change resulting from each individual action, and the equations can then be used to reduce these expressions and the ones that are built up from a sequence of actions. Table 1 sets out the results for the above sequence, showing the state changes that result from each action, to establish that the input value x_i is finally delivered correctly.

Table 1 State changes for a sequence of actions that operates correctly

	States				
Action	*[Sender]*	*[M1]*	*[Receiver]*	*[M2]*	*Notes*
ToS	$(1,x_i),(0,\varepsilon)$	-	-	-	new bit = 1
S_1	-	$(1,x_i)$	-	-	but is lost
timeout	-	-	-	-	no effect on data state
S_1	-	$(1,x_i)$	-	-	
S_2	-	-	$c_k(1,x_i),v$	-	new bit = 1
Deliver	-	-	-	-	output previous value of v
R_1	-	-	-	$c_k(1,x_i)$	
R_2	$(1,x_i),d_j(c_k(1,x_i))$	-	-	-	new bit = 1
ToS	$(0,x_{i+1}),(1,\varepsilon)$	-	-	-	
S_1	-	$(0,x_{i+1})$	-	-	
S_2	-	-	$c_{k+1}(0,x_{i+1}),c_k(1,x_i)$	-	
Deliver	-	-	-	-	output value$(c_k(1,x_i))$

Here, the significance of the bit changes is that the first S_2 action reestablishes one part of the invariant, and then the R_2 action reestablishes the second part. As the sequence continues after this R_2 action with a ToS (ie another loop of OneCycle), we must have value(States[Sender,2]) = value(States[Sender,1]): otherwise further iterations of Loop1 would occur until this condition was satisfied. The condition itself reduces to value$(d_j(c_k(1,x_i)))$ = x_i, and as d_j can not reverse a corruption produced by c_k we must therefore have value$(c_k(1,x_i))$ = x_i. This guarantees that x_i is output unchanged from the Receiver, which is the property that we wished to show.

7 ANALYSIS OF FAULTY OPERATION

The faulty operation can only occur in the case where a timeout causes parallel activity to occur, and this is illustrated by the sequence of actions shown in Figure 4, where the values of the bit associated with the message are shown above each action, along with an M or D to indicate whether the action is part of the Main or Duplicate sub-sequence respectively.

The basic operation of this sequence can be described informally as follows. It begins with x_i being accepted by the Sender and sent, not being corrupted by c_1, and being received, resulting in a Deliver that outputs x_{i-1}. A timeout then occurs, causing another copy of x_i to be sent. This is corrupted by c_2, but when it reaches the Receiver it will be stored, replacing the previous copy. Meanwhile, the correct copy will have been received back at the Sender, but has not been corrupted by d_1. The Sender will therefore accept a new value x_{i+1}, will flip the bit, and so will ignore the corrupted version (because it has the wrong bit value) when it is returned. Thus, when the message containing x_{i+1} is received, with its bit different to the one sent with x_i, the currently stored value of x_i will be delivered, even though it is corrupt.

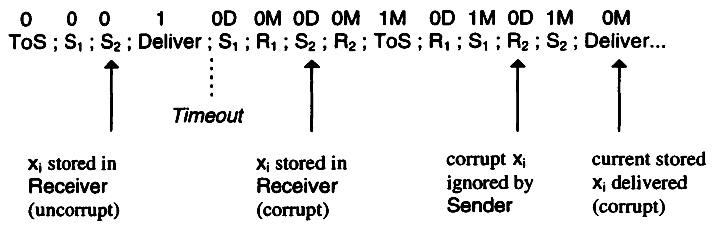

0 0 0 1 0D 0M 0D 0M 1M 0D 1M 0D 1M 0M
ToS ; S1 ; S2 ; Deliver ; S1 ; R1 ; S2 ; R2 ; ToS ; R1 ; S1 ; R2 ; S2 ; Deliver...
Timeout
xi stored in Receiver (uncorrupt)
xi stored in Receiver (corrupt)
corrupt xi ignored by Sender
current stored xi delivered (corrupt)

Figure 4 A sequence of actions that operates incorrectly

This can be analysed formally by working through the sequence of attributes to the actions in the same way as was done above for the simple cyclic operation. The results of this are presented in Table 2.

Table 2 State changes for a sequence of actions that operates incorrectly

	States				
Action	*[Sender]*	*[M1]*	*[Receiver]*	*[M2]*	*Notes*
ToS	$(1,x_i),(0,\varepsilon)$	-	-	-	
S_1	-	$(1,x_i)$	-	-	
S_2	-	-	$c_1(1,x_i),(0,x_{i-1})$	-	c_1 does not corrupt
Deliver	-	-	-	-	x_{i-1} is output
S_1	-	$(1,x_i)$	-	-	
R_1	-	-	-	$c_1(1,x_i)$	
S_2	-	-	$c_2(1,x_i),(0,x_{i-1})$	-	c_2 does corrupt
R_2	$(1,x_i),d_1(c_1(1,x_i))$	-	-	-	d_1 does not corrupt
ToS	$(0,x_{i+1}),(1,\varepsilon)$	-	-	-	
R_1	-	-	-	$c_2(1,x_i)$	
S_1	-	$(0,x_{i+1})$	-	-	
R_2	unchanged	-	-	-	input has wrong bit
S_2	-	-	$c_3(0,x_{i+1}),c_2(1,x_i)$	-	
Deliver	-	-	-	-	corrupt x_i output

8 SUMMARY AND CONCLUSIONS

We have therefore been able to demonstrate that the dataflow algebra model of this version of the alternating bit protocol is sufficiently powerful to enable the overall correctness of the system to be analysed, both in cases where it is guaranteed to function correctly and cases where it will fail.

These results are not particularly significant for the protocol (where they merely illustrate why setting timeout periods in protocols is a tricky problem for communications engineers), as most practical protocols now use some form of check bits rather than echo checking, and a version of this protocol that used check bits would not fail in the way that has been analysed here. What is important about these results is that, once the syntactic specification has been built for the whole system and the functional specifications constructed for the individual processes, then not only is their synthesis into the semantic specification almost mechanical, but so too is the analysis of it.

This simplicity is not because of any particular properties of the underlying computational model, as it appears that this should be equivalent to that of a value-passing process algebra such as the one used in (Bezem and Groote, 1994), although there is still much more work needed to explore the relationship between the dataflow algebra and process algebra approaches. Rather, the dataflow algebra approach gains its simplicity by making explicit the underlying chains of cause and effect within the processing that are implemented by the passing of messages between processes, whereas in a process algebra approach these chains need to be inferred (if possible) from patterns of behaviour that are implicit within the action trees generated by the algebra. In the dataflow algebra approach these explicit chains of causality are reflected in the essentially linear nature of the sequences, which then makes straightforward the derivation and reduction of the expressions for the state changes. Also, reasoning about the properties of the system is guided by the occurrence of loop structures in the sequences, as these highlight points where suitable invariants and stopping conditions need to be defined over the states of the processes.

Making explicit these chains of causality should also have a practical benefit, in that experience of the process of designing parallel and distributed systems suggests that designers will usually have at least an intuitive idea of the pattern of causality that they want to create. Indeed, some design approaches place considerable emphasis on describing the input-output behaviour of systems in terms of such patterns: for example the "use cases" of the ObjectOry method (Jacobson *et al*, 1992). Providing a specification methodology which allows designers to capture this intuition within a formal notation, so that it can be reasoned about, should therefore assist in formalising the development of such systems.

To be of maximum benefit, of course, this approach will need to be related more firmly to the models used to capture the behaviour of individual processes, such as process algebra models, but this still requires further investigation, along with the issue of how best to express the sort of relationships between the semantic and syntactic levels of specification that are important to constraining the interleavings generated by parallel composition of sequences of actions. Despite this, though, we believe that this example has demonstrated the validity of the dataflow algebra approach, and its potential value as a model for supporting the process of reasoning about the correctness of the overall behaviour of parallel and distributed systems.

9 ACKNOWLEDGEMENTS

The germ of the original idea behind this work was provided by George Wilson, and its development was greatly assisted by discussions with colleagues in the parallel processing research group, and notably Jon Kerridge, about their approaches to designing concurrent systems. The work done by Dave Cash in developing an early version of the notation used here for the syntactic level of the specification is also acknowledged.

10 REFERENCES

Bartlett, K., Scantlebury, R. and Wilkinson, W. (1969) A Note on Reliable Full-duplex Transmission over Half-duplex Links. *Communications of the ACM*, **12**, 260-261.

Bezem, M.A. and Groote, J.F. (1994) A Correctness Proof of a One-bit Sliding Window Protocol in μCRL. *Computer Journal*, **37**, 289-307.

C.C.T.A. (1990) *SSADM version 4 reference manual*. NCC Blackwell, Oxford.

Cowling, A.J. (1995) *Dataflow Algebras as Formal Specifications of Data Flows*. University of Sheffield Department of Computer Science Research Report **CS-95-18**.

Goguen, J.A. and Winkler, T (1988) *Introducing OBJ3*. Technical Report **SRI-CSL-88-9**, SRI International, Menlo Park, CA.

Goguen, J.A. (1991) Types as Theories, in *Topology and Category Theory in Computer Science* (eds. G.M. Reed, A.W. Roscoe and R.F. Wachter), Oxford University Press, Oxford.

Hoare, C.A.R. (1985) *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ.

Jacobson, I., Christerson, M., Jonsson, P. and Övergaard, G. (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Wokingham.

Kastens, U. (1991) Attribute Grammars as a Specification Method, in *Attribute Grammars, Applications and Systems* (eds. H. Alblas and B. Melichar), Lecture Notes in Computer Science **545**, Springer-Verlag, Berlin.

Knuth, D.E. (1968) Semantics of Context-Free languages. *Mathematical Systems Theory*, **2**, 127-145.

Milner, R. (1983) *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, NJ.

Yourdon, E. (1989) *Modern Structured Analysis*. Prentice Hall, Englewood Cliffs, NJ.

11 BIOGRAPHIES

Tony Cowling obtained a BSc with Honours in Computational Science and Mathematics in 1970 from the University of Leeds, followed by a PhD in 1977. Since 1973 he has been a lecturer in the Department of Computer Science at the University of Sheffield, with research interests in the formal modelling of software systems and in the teaching of software engineering.

Martin Nike obtained a BSc with Honours in Physics and Computing from Coventry University in 1992. He then gained an MSc in Parallel Computing and Computation from the University of Warwick in 1994. He is currently studying for a PhD at the University of Sheffield.

18
A hierarchical classification of overheads in parallel programs

J. M. Bull
University of Manchester
Centre for Novel Computing, Department of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK.
Telephone: 0161-2756144. Fax: 0161-2756204.
email: `markb@cs.man.ac.uk`

Abstract
Overhead analysis is a powerful conceptual tool in understanding the performance behaviour of parallel programs. Identifying the source of overheads in the program requires the classification of overheads into different types. We present hierarchical classification schemes for both temporal and spatial overheads, which facilitate the analysis of parallel programs on a wide variety of parallel architectures by providing a flexible framework for describing performance behaviour. The issue of measuring the various classes of overhead is also addressed, with a view to improving the utility of performance analysis tools.

Keywords
Parallel programs, temporal overheads, spatial overheads, classification

1 INTRODUCTION

Performance tuning of programs is a key activity undertaken by users of parallel computers. An important part of this process consists of reducing the overheads associated with parallelism (as opposed to tuning the single processor performance of the parallel code). These overheads are of two types–**temporal** and **spatial**. Reducing temporal overheads assists the programmer in obtaining an acceptable execution time for the program in question, while reducing spatial overheads allows the program to make acceptable demands on memory. What is 'acceptable' will, of course, depend on the intended use of the program. At any stage in the process of performance tuning a program, it is useful for the programmer to know the source of the observed overheads. The purpose of this paper is to present classification schemes for both temporal and spatial overheads that encompass all sources. In contrast to previously proposed schemes, (see, for example, Burkhart and Millen (1989), Crovella and LeBlanc (1994), Eigenmann (1993), Tsuei and Vernon (1990) and Vrsalovic *et al.* (1988)) ours is hierarchical. The motivation for this is that the importance of any given class of overhead can vary considerably, depending on the program and architecture being considered. Thus any single-level classification will either omit some important detail, or include unnecessary complexity for some program/architecture combinations. A hierarchical classification can be extended in areas of interest, or truncated in classes that are negligible, for the particular problem being considered. Thus it hoped we can provide a more flexible framework in which the programmer can reason about the performance of a program.

As discussed in Crovella and LeBlanc (1994), any classification scheme should have the following properties–

- **Completeness.** Any source of overhead should be classifiable within the scheme.
- **Orthogonality.** No source of overhead should appear in two different categories, unless one of the categories is a subset of the other.
- **Meaningfulness.** The classification should be meaningful, useful, and widely applicable in the context for which it is designed.

Crovella and LeBlanc acknowledge that their classification scheme is incomplete–we aim to produce a complete scheme. Other schemes such as those in Anderson and Lazowska (1990) and Martenosi *et al.* (1992) are also focussed on certain types of overhead. Unlike the first two properties, which can be objectively tested, meaningfulness is a subjective criterion. There is no one overhead classification scheme which can be said to be correct. In order to be able to assess the meaningfulness of our classification scheme, we must clearly state the context we have in mind:

Our primary purpose is to assist the programmer in choosing suitable code modifications to enhance the performance of a program on a given architecture.

In particular the scheme is not designed to assist the computer architect in modifying the hardware so that a given program will run faster. Thus the categories that we choose will reflect abstract programming concepts at the highest levels, with hardware features only appearing near the leaves of the hierarchy, if at all. By designing our hierarchy in this way, architectural differences can be incorporated by relatively minor changes to the scheme, making our scheme applicable to a wide range of parallel machines. It is therefore intended that ours should be a program-oriented scheme, rather than a hardware-oriented scheme such as those described in Burkhart and Millen (1989), Tsuei and Vernon (1990), and to some extent in Crovella and LeBlanc (1994) where hardware resource contention is given as a primary category of overhead. In the process of performance tuning, a programmer can be greatly assisted by the use of performance analysis tools. We believe that a hierarchical classification scheme could be usefully employed by such a tool as a means of clearly explaining to the programmer the reasons for the temporal and spatial behaviour of the program.

Another property we could have added to our list is measurability. We might like our classes of overhead to be easily measurable on a given architecture. However, this property tends to conflict with all three properties listed above. In most real systems, not all overhead sources are easily measurable, measurement can result in counting some overheads twice, and what is easily measurable may not be meaningful. It is clear that measurability is the major criterion which drives the definition and classification of overheads for the Cedar system presented in Eigenmann (1993), and that this both restricts the meaningfulness and applicability of the classification. We will take the opposite view, in the sense that we will avoid measurability as a criterion for defining classes of overheads. Nevertheless measurability is an important issue and we will return to it later.

2 TEMPORAL OVERHEADS

2.1 Definition

Before we can proceed to a classification scheme, we need to define what we mean by temporal overheads. Intuitively we would like the definition of temporal overheads to reflect the difference between the observed execution time of a program and the best execution time we could possibly expect if the entire program were parallelisable, and parallelism were free. Unfortunately, obtaining a tight lower bound on the execution time of a program is normally very difficult, since increasing the number of processors nearly always implies increasing the amount of fast memory (registers, vector registers or cache) available, resulting in the possibility of so-called superlinear speedup (see Helmbold and McDowell (1989)). The complexity of behaviour of fast memory (particularly cache) means that accurate simulation is the only way to obtain tight bounds on execution time. Thus we must accept that this ideal situation is in practice unattainable, and it makes sense to compare the observed execution time with a simple estimate of the 'best possible' execution time, bearing in mind that it will not in general be a lower bound. The simplest estimate we can give is to divide the execution time of a sequential version of the code by the number of processors. We define the temporal overhead on p processors O_p^T as

$$O_p^T = T_p - \frac{T_s}{p}, \tag{1}$$

where T_p is the observed execution time of the parallel code on p processors, and T_s is the execution time of the sequential code. Note that we will consider different classes of overhead as accounting separately for a certain amount of execution time and hence the classes will contribute additively to O_p^T. This means that they do **not** contribute multiplicatively to speedup or efficiency. This property also assists in verifying the completeness and orthogonality of a set of overhead measurements for a particular program, since the sum of the overheads over all classes should equal the total overhead directly measured by the above formulae. For the purposes of this study, we will assume that each processor runs a single thread of control.

Obviously there is the question of which sequential code is used to obtain T_s. This choice can be left to the programmer, as it will depend on what versions are readily available. On one hand, the programmer might wish to use a highly optimised implementation of a good sequential algorithm, in which case the overhead will include any algorithmic changes introduced to facilitate parallelism. On the other hand it may be more convenient to use a one processor version of the parallel code, but the programmer must be aware of the overheads that are being ignored by so doing. Frequently it will be some intermediate version between these extremes which is used.

It is tempting to divide overheads into those incurred by going from the best sequential version to a one processor parallel version and those incurred by going from the one processor parallel version to the many processor parallel version. This presents some difficulties, however, since some useful classes of overhead such as synchronisation, scheduling and data access may be split between these two classes, whereas they are each better understood as a single class. In addition, there are cases where the parallel algorithm makes no sense for a single processor.

There are cases where T_s is unobtainable because a single processor does not have access to enough memory to run the problem, or else the execution time of the sequential version is unacceptably long. In these cases it may be necessary to extrapolate T_s from problem sizes which can be run, or else to use the approaches described in Crovella and LeBlanc (1994) and Hockney (1995), doing the entire overhead analysis on smaller problem sizes, and using modelling techniques to extrapolate the results to larger problem sizes. Neither approach is very satisfactory, however, as many overhead sources are difficult to model using simple functions of program parameters.

2.2 Classification Scheme

The main structure of the hierarchical classification scheme is shown in Figure 1. At the highest level, we identify four classes of temporal overhead–

T1 *Information movement.* Overheads associated with the movement of information in the system.
T2 *Critical path.* Overheads resulting from the critical computation path through the parallel program being longer than the ideal.
T3 *Control of parallelism.* Overheads resulting from additional code which does not contribute directly to the solution of the problem, but serves to manage the parallelism.
T4 *Additional computation.* Overheads resulting from changes to the sequential code in order to increase parallelism requiring more computation to be executed.

Information movement

Information movement can be split into two classes, depending on whether the information being moved is data associated with the program, or information about the state of the computation–

T1.1 *Data accesses.* Additional time spent making data accesses. Note that any time spent making data accesses which is overlapped with computation is not considered as an overhead.
T1.2 *Synchronisation.* Time spent in performing synchronisation operations.

The data access overhead can be negative, owing to the increase in the amount of fast memory available as the number of processors is increased. Data access overheads can themselves be split into classes, according the different possible paths between levels of the memory hierarchy, e.g. local memory to local cache, local memory to registers (bypassing cache), remote memory to local memory, disk to local memory. The number and nature of these classes will depend on the architecture concerned. Note that any changes to the code to increase parallelism may result in a different amount of time being spent in data accesses, thus contributing to this class. In many parallel programs on distributed

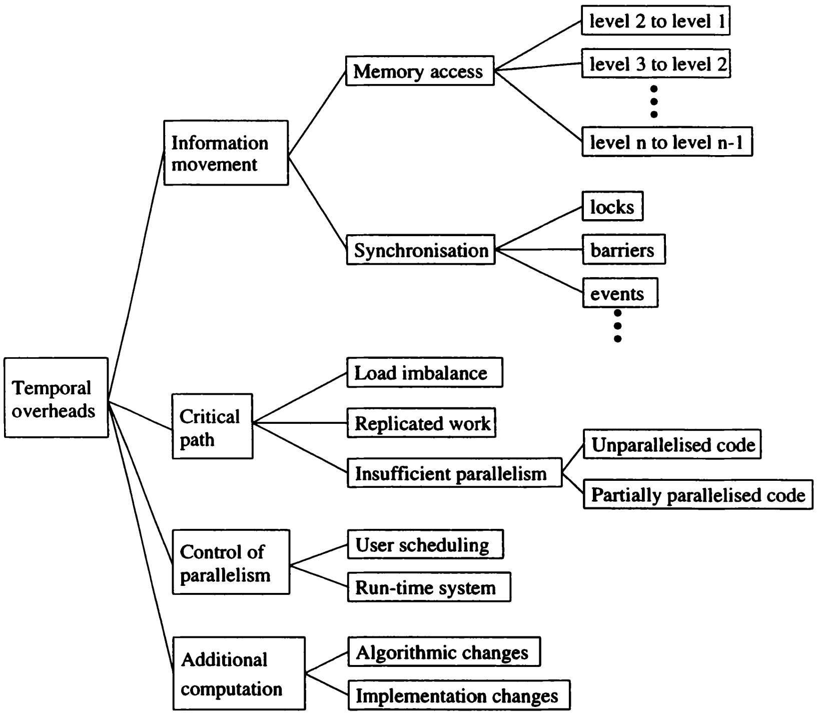


Figure 1 Classification of temporal overheads.

memory machines the greatest contribution to data access overheads will be from the remote memory to local memory path. It may prove useful to further subdivide this class– for example to distinguish between latency and bandwidth effects in the interconnection network, in message passing architectures to distinguish between read and writes, or in virtual shared memory architectures to distinguish between capacity misses and coherency misses. In contrast, data access overheads tend to be less significant on true shared memory architectures.

Synchronisation overheads may be subdivided according to the type of synchronisation construct being employed. The most commonly used are barriers, locks and events, though many other types do exist. It should be stressed that in this classification, synchronisation overheads **do not** include time spent waiting at synchronisation points. This time is accounted for by the class *T2* (critical path).

Critical path

Critical path overheads arise from imperfect parallelisation of the program. The usual effect of this is that processors will be idle, waiting at a synchronisation point for some period of time. Another possibility is that many processors may be executing the same code, since this may be less expensive that executing it on one processor, and then incurring data access overheads in distributing the results. We divide the critical path overheads into three classes–

T2.1 *Load imbalance.* Time spent waiting at a synchronisation point, because, although there are sufficient parallel tasks, they are asymmetric in the time taken to execute them.
T2.2 *Replicated work.* Processors are not idle, but are occupied in replicating computation which is executed on other processors.
T2.3 *Insufficient parallelism.* Processors are idle because there is an insufficient number of parallel tasks available for execution at that stage of the program.

Insufficient parallelism has a variety of causes, giving rise to subdivisions of this class–

T2.3.1 *Unparallelised code.* Time spent waiting in sections of code where there is a single task, run on a single processor.
T2.3.2 *Partially parallelised code.* Time spent waiting in sections of code where there is more than one task, but not enough to keep all processors active.

Unparallelised code can occur because there exist data dependencies preventing its parallelisation, because it would more costly if it were parallelised, or simply because the programmer has not yet addressed the parallelisation of the code section. Partial parallelism can arise in a variety of parallel constructs, including fan-in/fan-out operations, critical sections, DOACROSS loops and wavefront loops. Note that distributed operating system code is a source of insufficient parallelism overhead, since access to shared resources such as page tables and I/O channels may be sequentialised.

Management of parallelism

This class of overheads arises because code which does not contribute to the generation of a solution, but is added to the sequential version to control and manage parallel tasks. The time spent in this activity may include executing instructions, data accesses and synchronisation, but it is not especially useful to sub-classify in this way. It is more meaningful for the programmer to divide it as follows–

T3.1 *User scheduling.* Time spent in user-written code that controls what parallel tasks are executed on which processor and when.
T3.2 *Run-time system.* Time spent in run-time system code.

Additional computation

In Section 2.1 we allowed the programmer to choose exactly how T_s is determined, and promised to include overheads resulting from changes to the sequential code in order to increase the available parallelism. So far we have accounted for additional data accesses arising from this source, but it remains to account for additional computation which may result. We sub-classify this overhead according to the level of specification at which the changes occur. As discussed in Gurd *et al.* (1993), levels of specification can be defined arbitrarily, so we are free to choose the most useful. We choose (without being rigorous) a specification level similar to the 'algorithm' level described in Gurd *et al.* (1993), which states what is to be computed in terms of simple arithmetic operations, basic mathematical functions, indexed variables and simple set/logic concepts. Notice that such a description contains no information about how data is organised, nor any ordering of computation other than that required by the algorithm's data dependency structure.

We can classify additional computation according to whether it is caused by changes above or below this specification level–

T4.1 Algorithmic changes. Time spent in additional computation, resulting from changes at this specification level.

T4.2 Implementation changes. Time spent in additional computation, resulting from changes below this specification level.

Implementation changes include all the transformations that a restructuring compiler might make (loop interchange or fusion, for example), but also transformations that make use of some simple mathematical equivalences.

3 SPATIAL OVERHEADS

3.1 Definition

Before we can define spatial overheads, we must define more carefully what we mean by memory requirements. We choose the **maximum instantaneous memory usage**, since this the key quantity if there is a hard limit on the amount of memory available. It also has no temporal component, thus avoiding any overlap with temporal overheads. We would like spatial overheads to represent the difference between the memory requirements of a parallel code and those of its sequential counterpart. Our task is somewhat easier than for the case of temporal overheads, since the maximum instantaneous memory usage of the sequential program M_s is a reasonable lower bound on the maximum instantaneous memory usage of the parallel program M_p. We can therefore define the spatial overhead O_p^S as

$$O_p^S = M_p - M_s. \tag{2}$$

Once again there is the question of which version we use to measure M_s. If we wish to consider trade-offs between temporal and spatial overheads, then clearly we should use the same code used to measure T_s. Otherwise we allow the programmer to make their own choice, again bearing in mind that some overheads will not be accounted for if the one processor parallel version is used. Again, it is possible to split sequential to one-processor-parallel overheads into a separate class, but similar arguments to those used in the temporal case can be applied to justify avoiding this. In situations where M_s is not measurable, because the sequential code does not have access to enough memory, or will run for unacceptably long, it will be necessary to extrapolate M_s from smaller problem sizes. This is normally a much easier task than extrapolating T_s, as there is often a simple relationship between problem size and memory usage.

3.2 Classification scheme

Figure 2 shows the hierarchical classification scheme for spatial overheads. The primary

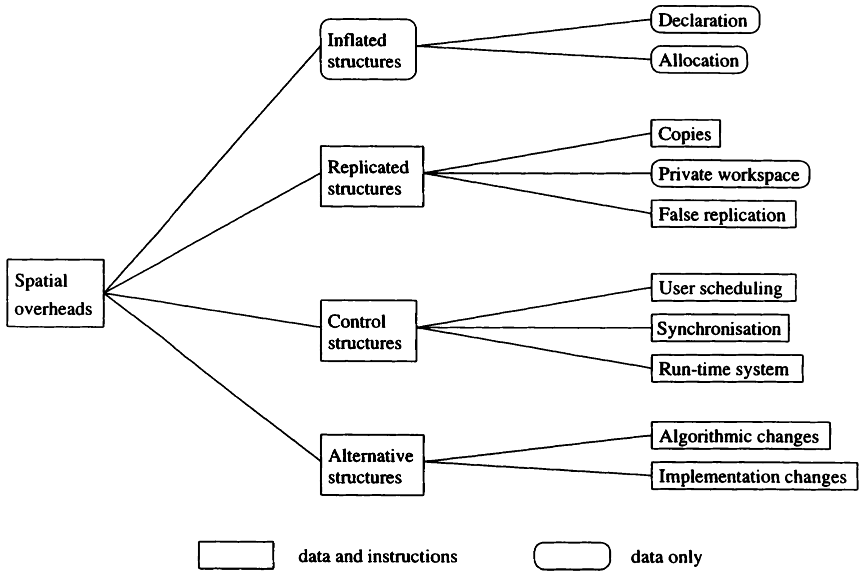


Figure 2 Classification of spatial overheads.

distinction in spatial overheads is between program data and instructions. However, since the subtree for instructions is a subset of the subtree for data, only the classification for data is shown, with those nodes which do not apply to instructions indicated.

At the next level we consider four main types of spatial overhead–

S1 Inflated structures. Additional memory required so that some dimension of a data structure is a convenient size.

S2 Replicated structures. Additional memory required because multiple versions of a data structure exist in the parallel code.

S3 Control structures. Additional memory required for data structures which are added to the program to enable the control and management of parallel tasks.

S4 Alternative structures. Additional memory resulting from changes to the sequential code in order to increase parallelism.

Inflated structures

Inflation of data structures only occurs in program data and not in instructions. Inflation takes a number of forms, but we can distinguish between two subclasses–

S1.1 Declaration. Inflation which occurs because the declared size of the data structure in the program has increased.

S1.2 Allocation. Inflation which occurs because memory allocation occurs in pages.

Declared inflation may occur because shared data structures are constrained to have certain sizes (for example, array dimensions must be powers of two on some systems). This effect is also seen on virtual shared memory machines, since a common technique to reduce false sharing is to pad the inner dimension of a data structure to the size of the coherency unit.

To understand allocated inflation, note that some space will be wasted on a single processor because the size of the data segment will be rounded up to a whole number of pages. On multiple processors, however, more memory may be wasted if each processor rounds up its segment to a whole number. This effect is compounded by any distinction between different types of data, as each type may have its own segment.

Replicated structures

Replication of data structures is often the most significant source of spatial overhead in parallel programs. We can classify replicated data by the way it is used in the program–

S2.1 Copies. Data space that is replicated and used to store copies of data stored elsewhere in memory.
S2.2 Private workspace. Data space that is replicated and used to store data values that are not replicated anywhere in the system.
S2.3 False replication. Declared data space that is replicated but never used.

Private workspace replication will occur on any architecture, and is the main source of replication on true shared memory machines. Copy replication is normally restricted to distributed memory, as its primary function is to avoid memory latency. Communication buffers are also a form of copy replication. Replication of instructions is limited to copy replication, and false replication. Note that false replication differs from allocated inflation in that in the former case the wasted memory corresponds to program variables whereas in the latter it does not. False replication occurs frequently in cache-only virtual shared memory systems, since space is allocated for a whole page even if the requesting processor only requires read access to a single word on that page. It may also occur in distributed memory systems when entire data structures are replicated because it is not known at compile time what subset of the structure will be accessed by a given processor.

Control structures

Control structures can be sub-classified according to the type of control–

S3.1 User scheduling. Data and instructions used by user-written code that controls what parallel tasks are executed on which processor and when.
S3.2 Synchronisation. Data and instructions associated with synchronisation constructs.
S3.3 Run-time system. Data and instructions associated with the run-time system.

Control structures are not frequently a major source of overhead, but they can sometimes be quite large; arrays of lock variables and arrays that describe task to processor mapping, for example.

Alternative structures

This class arises for precisely the same reasons as class *T4*–we must account for changes in memory requirements resulting from changes to the sequential code in order to increase the available parallelism. These overheads can be sub-classified in the same way as class *T4*-

S4.1 *Algorithmic changes.* Additional data and instructions resulting from changes at the algorithm specification level.

S4.2 *Implementation changes.* Additional data and instructions resulting from changes below this specification level.

It is possible that this class of overheads could be negative, if the changes result in lower storage costs but possibly longer (sequential) execution times.

4 MEASUREMENT

4.1 Temporal overheads

The most obvious requirement for measurement of temporal overheads is a direct consequence of the definition we have used: it is necessary to analyse the performance of both the parallel and sequential version of the code, and, to enable overhead localisation, to be able to correlate the corresponding regions of the code in the two versions. This is relatively easy to do at the subprogram level, but at a lower level may require explicit marking of basic code blocks.

Accurate measurement of temporal overheads imposes a number of requirements on hardware. At the very least each processor must have a high-resolution clock which can be read very cheaply. To record memory access times, hardware support is required for logging and timing misses in all the levels of the memory hierarchy, since most memory operations are too fine-grained to be measured in software.

Since both critical path and information movement overheads are often characterised by the CPU idling (or spin-waiting), it can be difficult to distinguish between the two. In a message-passing environment, the cost of communication between processors requires co-operation between the sending and receiving processes. The receiving process can time how long is spent in a blocking receive, but without knowing the time at which the message is sent, it cannot distinguish as to whether this is time is waiting for the message to arrive (memory access overhead), or waiting for the sending processor to finish some computation (load imbalance overhead). It is therefore necessary to have the send time available, either at run-time if statistics are being accumulated, or from any trace which is used for post run-time analysis. Some method of synchronising clocks between processors is also required.

Measuring synchronisation overheads poses some similar difficulties. A processor requesting a lock can tell how long it takes to acquire a lock, but it cannot divide this time into time spent waiting for the lock to be released by another processor (partially parallelised code overhead) and the time to transfer the lock (lock synchronisation overhead).

Control of parallelism, additional computation and replicated work overheads can only be identified by their program context. This is easy if the code responsible is packaged into subprograms (a run-time library for example), but much more difficult if the code is inlined. Again, some marking of code blocks may be required. The ability to count instructions (either in hardware or software) can also be of assistance here.

4.2 Spatial overheads

Measuring spatial overheads is significantly more difficult than measuring temporal overheads, since determining the amount of memory used by a program, especially in a virtual memory system, is in general harder than measuring the program's execution time. This may account for the general lack of memory statistics available from performance analysis tools.

Distinguishing between data and instructions, or between private and shared data is straightforward, since they occupy different segments of memory. Partial information about spatial overheads can be obtained by examining the size of statically declared data structures and tracking dynamic memory allocation–this will normally give an accurate picture of control structure overheads, for example. This approach will be in error, however, if structures are declared (or allocated) but not used in their entirety. Automatically tracking use of memory is not straightforward, but the programmer will often have available some knowledge of array access patterns which allows a good estimate of spatial overheads to be made. In contrast to temporal overheads, rough estimates may be adequate to allow the programmer to make the appropriate modifications to the code to sufficiently reduce spatial overheads. Usage of memory at the page level can be monitored by the operating system, by keeping track of page faults, but deciding which structures are being accessed either requires the use of symbol table information, or the use of different memory segments for different data structures. The problem is especially difficult in a virtual shared memory system, where multiple copies of pages can exist. Obtaining information about usage of memory at a lower level (e.g. how many words on a page are actually accessed) can only be achieved via address tracing which may be impractical for large programs. Distinguishing between copy overheads and false replication in a virtual shared memory system is an example of such a problem.

5 CONCLUSIONS AND FUTURE WORK

We have presented hierarchical classification schemes for both temporal and spatial overheads in parallel programs. Since the classification has been motivated by meaningfulness and usefulness for a programmer, we hope that the schemes will form a useful framework in which overheads can be analysed on a wide variety of parallel architectures.

Prototypes of the temporal scheme have already been used in the development of parallel programs (see Egan *et al.* (1994) and Falcó Korn *et al.* (1995) for examples), and it is intended to utilise the full schemes in future porting exercises, incorporating them into a general methodology for parallel program development. It is also intended that this type of framework be extended to multi-threaded architectures, which we specifically excluded here.

REFERENCES

Anderson, T.E. and Lazowska, E.D. (1990) Quartz: a tool for tuning parallel program performance, in *Proceedings of ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 115–125.

Burkhart, H. and Millen, R. (1989) Performance-measurement tools in a multiprocessor environment. *IEEE Trans. on Computers*, **38(5)**, 725–737.

Crovella, M.E. and LeBlanc, T.J. (1994) Parallel performance prediction using lost cycles analysis, in *Proceedings of Supercomputing '94*, IEEE Computer Society.

Egan G.K., Riley, G.D. and Bull, J.M. (1994) Parallelisation of the SDEM distinct element stress analysis code on the KSR-1. *Proceedings of ACM International Conf. on Supercomputing*, 85–92.

Eigenmann, R. (1993) Toward a methodology of optimizing programs for high performance computers. *Proceedings of ACM International Conference on Supercomputing*, 27–36.

Falcó Korn C., Bull J.M., Riley G.D. and Stansby P.K., (1995) Parallelisation of a three-dimensional shallow water estuary model on the KSR-1. *Scientific Programming*, **4(3)**, 155–170.

Gurd, J.R., Cooper M.D., Hedayat G.A., Nisbet, A., O'Boyle, M.F.P., Snelling D.F. and Böhm, A.P.W. (1993) A framework for experimental analysis of parallel computing. *University of Manchester Department of Computer Science Technical Report UMCS-93-2-3*, University of Manchester, UK.

Helmbold, D.P. and McDowell, C.E. (1989) Modeling speedup(n) greater than n, in *Proceedings of 1989 Int. Conf. on Parallel Processing*, Pennsylvania State Univ. Press, III-219–III-225.

Hockney, R.W. (1995) Computational similarity. To appear in *Concurrency: Practice and Experience.*

Martenosi, M., Gupta, A. and Anderson, T. (1992) MemSpy: analyzing memory system bottlenecks in programs, in *Proceedings of ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1–12.

Tsuei, T.-F. and Vernon, M.K. (1990) Diagnosing parallel program speedup limitations using resource contention models, in *Proceedings of 1990 Int. Conf. on Parallel Processing*, Pennsylvania State Univ. Press, I-185–I-189.

Vrsalovic, D., Siewiorek, D.P., Segal, Z.Z. and Gehringer, E.F. (1988) Performance prediction and calibration for a class of multiprocessor systems. *IEEE Trans. on Computers* **37**, 1353–1365.

BIOGRAPHY

Mark Bull received the M.A. degree in Mathematics from the University of Cambridge, and the M.Sc. degree in Numerical Analysis from the University of Manchester. He is currently a Research Associate in the Centre for Novel Computing within the Department of Computer Science at the University of Manchester. His main research interests are in the design and implementation of parallel numerical algorithms and in parallel programming techniques and methodology, particularly for virtual shared memory architectures. Previously he worked as a researcher in atmospheric boundary layer simulation at the UK Meteorological Office.

19
Periodicity in an asynchronous algorithm for parallel processing

L. R. Fletcher and M. Santini
Department of Mathematics and Computer Science
University of Salford
Salford, Lancashire M5 4WT, United Kingdom
Tel: +44 (0) 161 745 5406, Fax: +44 (0) 161 745 5559
e-mail: L.R.Fletcher@mcs.salford.ac.uk, M.Santini@mcs.salford.ac.uk

Abstract

We consider asynchronous iterative algorithms for distributed processes in networks in which the computations at each node are allocated to a fixed process. Each process begins a new iteration when it has received new values from a 'sufficient' number of the processes to which it is adjacent in the dependency graph. For each process different 'sufficient' criteria might be chosen; for example at least 'one', a given subset or 'all' the adjacent processes. Each process uses only the most up-to-date of the values it has received, discarding any earlier values. We assume that each process performs an iteration in its own constant commensurable time and to each channel corresponds a fixed communication delay.

We show that the behaviour of the processes in the overall network is predictable as each process computes periodically. Moreover, if the dependency graph is strongly connected and 'sufficient' means 'all' the adjacent processes, the computation rate is the same for each process and there emerges a calculable formula for this rate. Thus the algorithm becomes implicitly synchronous and its performance can be evaluated.

Keywords

Asynchronous algorithms, parallel processing, distributed processes, dynamic behaviour, K-periodicity, performance, Petri net, timed Petri net

1 INTRODUCTION

We consider iterative algorithms of the form

$$x^{(k)} := f\left(x^{(k-1)}\right)$$

where $x = (x_1, x_2, \dots, x_n)$ is a vector in $\Re^n$ and $f : \Re^n \longrightarrow \Re^n$ is an iteration mapping defining the algorithm. These algorithms can be executed on a parallel or distributed computing system in which the i-th process Q_i updates x_i according to the formula

$$x_i := f_i(x_1, x_2, \dots, x_n).$$

We will need to be specific about communication links between processes so we define subsets $\mathcal{I}_1, \mathcal{I}_2, \dots, \mathcal{I}_n \subseteq \{1, 2, \dots, n\}$ by

$$j \in \mathcal{I}_i \Leftrightarrow f_i \text{ depends on } x_j,\ j \neq i.$$

The *dependency graph* $\mathcal{G}$ is the directed graph with n vertices, labelled $1, 2, \dots, n$, with the directed edge (i, j) present if and only if $j \in \mathcal{I}_i$. We will assume that the communication network between the processes is isomorphic to the dependency graph and that each process Q_i is aware of the current value of x_i.

We focus here on an asynchronous implementation, in which each process computes at its own pace while receiving information on the values of the components updated by the other processes. Descriptions of asynchronous iterations have been presented by different authors including (Baudet 1978) and (Bertsekas and Tsitsiklis 1991). We will work with a slightly modified version of Bertsekas's formulation.

Write $\mathcal{I}_i = \{i_1, i_2, \dots, i_{\alpha_i}\}$, $i = 1, 2, \dots, n$, where $\mathcal{I}_i$ is the communication set to Q_i defined above. Let $x_i(\theta)$ be the value of x_i, residing in the memory of the i-th process at time θ, where θ is a natural number. According to (Bertsekas and Tsitsiklis 1991) the asynchronous version of the iterative procedure is given by

$$\begin{cases} x_i(\theta) = f_i(x_{i_1}(\tau^i_{i_1}(\theta)), \dots, x_{i_{\alpha_i}}(\tau^i_{i_{\alpha_i}}(\theta))), & \text{if } \theta \in \Theta^i \\ x_i(\theta) = x_i(\theta - 1) & \text{otherwise} \end{cases}$$

where the schedule Θ^i is the set of times at which x_i is updated (that is, Q_i finishes a computation) and $\tau^i_j(\theta)$ are times satisfying

$$0 \leq \tau^i_j(\theta) \leq \theta - 1, \text{ for all } \theta \geq 1.$$

At time θ, $x_j(\tau^i_j(\theta))$ is the last value computed by Q_j that Q_i has received. We chose not to include i in I_i to simplify the notation. If f_i depends on x_i then the reader will easily verify that none of the results is affected, as process i is always aware of the current value of x_i.

We assume that each process Q_i, $i = 1, 2, \dots, n$ satisfies the following communication rule. As soon as a process Q_i finishes a computation it sends its result x_i to the Q_j which need it, according to the dependency graph. Q_i carries out a new iteration as soon as a 'sufficient' number of updated component values have been received from other processes. We denote by $\mathcal{H}_i \subseteq 2^{I_i}$ the 'sufficient' set of subsets of indices of values (or processes). As soon as Q_i has received enough new values whose indices make up an element of $\mathcal{H}_i$, process Q_i commences a new iteration. It is natural to require that, when this occurs, Q_i uses the most recently received value of every variable. Hence we impose the condition

$$H \in \mathcal{H}_i \quad \Rightarrow \quad L \in \mathcal{H}_i \text{ for every } L \text{ such that } H \subseteq L \subseteq I_i \tag{1}$$

and require that process Q_i re-starts with the biggest element of $\mathcal{H}_i$ available. If two elements of $\mathcal{H}_i$ are available at the same time, by (1) their union is also an element of $\mathcal{H}_i$ and then chosen for the new computation. We discuss now a few particular cases of $\mathcal{H}_i$.

- If 'sufficient' means 'all' for each process, Q_i re-computes as soon as it has received a new value from each of the processes Q_h, $h \in I_i$. $\mathcal{H}_i$ has got a single element, $\mathcal{H}_i = \{I_i\}$. We analysed this case in detail in an earlier paper (Fletcher and Santini 1994).

- If 'sufficient' means 'one' for each process, without any particular requirement about which one, Q_i re-starts with any number of new values. So Q_i stays idle only if it received no communications since it began the previous iteration. In this case $\mathcal{H}_i = 2^{I_i}$.

- If 'sufficient' means that Q_i re-computes x_i only when it has received updated values from x_h for $h \in H_i \subseteq I_i$ then $\mathcal{H}_i = \{H_i \cup I \mid I \in 2^{I_i - H_i}\}$.

After each iteration Q_i picks up the most recent values available, and waits if necessary for other values depending on the criterion chosen. If, by the time it re-starts, more than one value from the same process has arrived, it uses only the latest value and discards the previous ones. Thus one process Q_i will buffer no more than Card H_i values at each iteration, that is, no more than one value per process it is supposed to receive from. The re-starting rule can be decomposed in three ordered subrules.

1. Each process re-starts as soon as the criterion is satisfied;

2. With all the new values available;

3. With the most up-to-date of those new values.

In any case Q_i may have to wait a certain period of time before re-computing, and it is more likely to be idle with the smaller the set $\mathcal{H}_i$ is, the extreme case being $\mathcal{H}_i = \{I_i\}$. At the other extreme, if $\mathcal{H}_i = 2^{I_i}$ then the amount of asynchronism is more significant and the idle time of the processes is reduced.

We can consider the receiving rule as a local synchronisation, whereas the sending of a message does not imply any synchronisation (the receiving process can be in a computing state). As soon as Q_i finishes an iteration, it becomes aware of the values received when it was computing. We assume that the communication system is reliable (messages neither lost nor corrupted nor desequenced) so the values are definitely received after being sent.

In order to have a clear picture of the processes' behaviour, we make the following assumptions.

1. The dependency graph of the processes is connected.
2. At time $\theta = 0$ all the processes are provided with the initial value of the components of x they need, according to the dependency graph causing them to commence their computation. We shall say that each process is *launched.*
3. Each Q_i computes in its own speed and the computation time d_i of Q_i is the same at all iterations. It is convenient to assume that these processing times are commensurable.
4. We denote by d_{ij} the time required to send a value in the channel linking Q_i to Q_j. We assume that the d_{ij} are constant and commensurable. As soon as Q_i finishes a computation, it sends its result to the Q_j which need it, according to the dependency graph. The Q_j receive it d_{ij} units of time later.

Such an algorithm will be called in the following a WS(Weakly Synchronized)-algorithm. The justification of the 'WS-algorithm' terminology lies in Theorem 1 where we show that from the local synchronisation emerges behaviour which mimics a globally synchronised algorithm. Therefore the behaviour of the processes is predictable rather than chaotic.

Theorem 1 states that the processes compute cyclically after a transient initial phase. In section 2 we give the main steps of the proof of Theorem 1 where the WS-algorithm is modeled by a timed priority system (timed Petri net with priorities) with the same dynamic behaviour. Section 3 deals with the particular criterion 'all', that is $\mathcal{H}_i = \{I_i\}$. We refer to (Carlier, Chretienne and Girault 1985) to show that the same computation rate emerges for each process, and give a calculable formula. Section 4 gives one example.

Related issues arise in the analysis of temporal properties of parallel compositions of omega-automata (Alur, Itai, Kurshan and Yannakakis 1995) and protocol analysis and verification (Aggarwal, Barbara and Meth 1987). However our work is motivated towards the analysis of asynchronous implementations of numerical algorithms and their performance, in relation to convergence for example (Bull and Freeman 1992).

2 BEHAVIOUR OF A WS-ALGORITHM

2.1 Definition of K-periodicity

The type of periodicity we are concerned with is called K-periodicity and was introduced by (Chretienne 1983). A schedule is a sequence whose elements are time occurences. Intuitively a schedule is K-periodic if the K succesive time occurences repeat periodically.

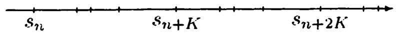


Figure 1. Example of a K-periodic sequence.

More formally we have the following definition in (Carlier and Chretienne 1988).

Definition 1 *A sequence $\{s_n\}$, $s_n \in \mathbb{Z}^+$, is K-periodic, with period π, if there exists an integer n_0 such that*

for all $n \geq n_0$, $s_{n+K} = s_n + \pi$

where $K \in \mathbb{Z}^+$ is the periodicity factor, π the period, and K/π the frequency of $\{s_n\}$.

2.2 K-periodicity in a WS-algorithm

In this section we show that after a transient phase each process in a WS-algorithm computes periodically. Thus the behaviour of each process – when it will be performing and when it will be idle – is predictable. This result is expressed in Theorem 1 for which we give an outline proof; full details appear in (Fletcher and Santini 1995).

Theorem 1 *For each process Q_j in an WS-algorithm there is a K_j such that Q_j computes under a K_j-periodic schedule.*

Proof.
The proof lies in the modelling of the WS-algorithm by a *timed priority system*, that is a timed Petri net with priorities (Best and Koutny 1992). The timed priority system has properties which apply to the underlying WS-algorithm to prove the theorem. In the following we shall assume that the reader is already familiar with Petri nets; we refer to (Murata 1989) for concepts and definitions relating to Petri nets.

Because each process computes for a fixed amount of time, it is natural to model one process of computing delay d_i by a d_i timed transition, the transition firing meaning that the process is computing. With any non-zero communication delay d_{jk} will be associated a timed transition of duration d_{jk}. A new value received is symbolised by a token in the input place; the exchange of data is represented by the token traffic.

Such a Petri net structure is analogous to the dependency graph of the processes. However, we are considering processes performing several tasks: computing, receiving, discarding and sending. Therefore we need a more elaborate Petri net model in which some new elements are added to the model of each process.

1. A process is one agent and cannot compute two values at the same time, whereas a timed transition can initiate a new firing even if it has not terminated the previous one. Thus, to model a process we want two firings of the associated timed transition not to overlap. To achieve this, with each timed transition we associate a single place loop.

2. In the WS-algorithm values are sometimes discarded. This happens when two values of x_i (or more) from Q_i are available at one process Q_j. Process Q_j picks up only the last one, which is supposed to be the most up-to-date. In terms of Petri nets we interpret it as two tokens available in one place. One has to be removed so we add transition and arcs to remove the extra tokens.

3. As Petri nets model conditions it is possible to model the re-starting criterion by adding places, transitions and arcs within the model of one single process. Thus the model will be different for different criteria. Those extra Petri net features bring unwanted conflicts in the structure. Which transition is going to fire if two are enabled at the same time? To keep a deterministic system, even if it is not structurally conflict free, a priority relation is introduced for the transitions of each subnet model of a process, resulting in a timed priority system.

4. The initial marking corresponds to the launching of the processes. The net without tokens is like the network with all the processes dead. As soon as the initial tokens are put in the net, the transitions start firing, according to the rule 'fire a transition as soon as it is enabled'. The dynamic behaviour of the Petri net will model the behaviour of the processes (idle or performing) in the WS-algorithm.

Remark. Those extra Petri net elements are only added inside the subnet model of each single process. This is necessary to keep a fully decentralised system of processes, without any extraneous interactions between the processes.

We have constructed a marked timed priority system whose transitions fire as soon as they are enabled and whose Petri net is bounded (Fletcher and Santini 1995). Moreover, after a certain time – depending on the topology of the network, the computation and communication times and the manner in which the overall process is launched – this Petri net has the same dynamic behaviour the live subnet of the original Petri net model. Then, using properties of the new live bounded timed priority system, we can derive a theorem showing that the times at which each transition begins to fire is a K-periodic schedule. Then we can transfer these results to the WS-algorithm and prove Theorem 1.

Using Bertsekas formulation Theorem 1 can be re-stated.

Corollary 1 *There exists a time θ_0 and integers π, K_j, $a_j^1, \ldots, a_j^{K_j}$, $j = 1, \ldots, n$ such that for all $\theta \geq \theta_0$,*

$$\Theta^j = a_j^l \bmod \pi, \; l \in \{1, \ldots, K_j\}$$

where π is the period and K_j the periodicity factor of process j.

3 CRITERION 'ALL': EVALUATION OF THE PERFORMANCE

We focus here on the particular criterion 'all', that is $H_j = I_j$ for all $j = 1, \ldots, n$. For processes in a strongly connected network we show that

$$K_1 = K_2 = \cdots = K_n = K$$

and derive a formula for the computation rate $f = K/\pi$.

Theorem 2 *Consider a WS-algorithm in a strongly connected network computing under the criterion 'all'. Then each process computes under a K-periodic schedule and the computation rate f is then given by the formula*

$$f = \min_k \left\{ \frac{1}{d_{max}}, \frac{\#\{Processes\ in\ C_k\} - \#\{Values\ discarded\ in\ C_k\}}{Total\ delay\ in\ C_k} \right\}$$

where d_{max} is the largest computation time for any process in the network and $C_1, \ldots, C_l$ are the simple circuits of the graph.

Proof.
A fully detailed proof of Theorem 2 is given elsewhere (Fletcher and Santini 1994, 1995). We develop here the important steps.

We refer to (Carlier, Chretienne and Girault 1985) where they derive a theorem for what they call the earliest controlled execution of a Petri net. Most importantly they give a formula calculating the frequency of the firings of each transition, having shown already that those firings are K-periodic. The earliest controlled execution is a particular execution of the firings of a Petri net, which for the subclass of Petri nets called *timed marked graph* follows the rule: 'Fire a transition as soon as enabled'. Firing as soon as enabled is one requirement of our timed priority system model of the WS-algorithm. However our Petri net model is generally not a timed marked graph.

With criterion 'all' there exists a time θ_0 such that, for all θ greater than θ_0, the timed Petri net model of the WS-algorithm is equivalent to a timed marked graph. In fact, the live subnet emerges as a timed marked graph. As there are no conflicts in a marked graph, the priority relation is no longer necessary.

Within the transient phase values may have been discarded so the initial marking of the Petri net may not be reachable from the marking at θ_0. To use the theorem from (Carlier *et al* 1985) we need the marking at θ_0. In terms of the WS-algorithm this requires the algorithm to be run up to time θ_0 to find out where values are discarded. With the particular criterion 'all' no values are discarded after the transient time has elapsed.

As the formula only works for strongly connected timed marked graphs, we restrict ourselves to strongly connected networks of processes. Then we can apply the theorem from (Carlier*et al* 1985) to our timed Petri net model of a WS-algorithm and prove Theorem 2.

This theorem gives the exact computation rate, but we need to run the algorithm up to the steady state. We can get an upper bound for the computation rate with the following obvious consequence of our earlier result.

Corollary 2 *Consider a WS-algorithm in a strongly connected network computing under the criterion 'all'. The computation rate f of the algorithm in its steady state satisfies*

$$f \leq \min_k \left\{ \frac{1}{d_{max}}, \frac{\#\{Processes\ in\ C_k\}}{Total\ delay\ in\ C_k} \right\}$$

where d_{max} is the largest computation time for any process in the network and $C_1, \ldots, C_l$ are the simple circuits of the graph.

We now use Bertsekas formulation of the WS-algorithm, to re-state the above results.

Corollary 3 *There exists a time θ_0 and the displacement times $a_i^1, \ldots, a_i^K$, $i = 1, \ldots, n$ such that for all $\theta \geq \theta_0$,*

$$\Theta^i = a_i^l \bmod \pi, \ l \in \{1, \ldots, K\}.$$

K and π are the smallest integers such that $f = K/\pi$, f defined as in Theorem 2.

Hence the schedule $\{\Theta^i | \theta \geq \theta_0\}$, $i = 1, \ldots, n$ is exactly predictable if the algorithm is run up to θ_0. As soon as the steady state is established each process computes K times in a period and the algorithm becomes synchronous.

4 EXAMPLE

We consider the strongly connected network of four processes illustrated in Figure 2. Process Q_2 is the only process of the graph receiving data from more than one other process. Although our analysis applies to any criterion satisfying (1), here we discuss the cases

$$\mathcal{H}_2 = \{(1), (4), (1,4)\},$$
$$\mathcal{H}_2 = \{(1,4)\}.$$

With $\mathcal{H}_2 = \{(1,4)\}$, that is criterion 'all', we apply Theorem 2 in order to calculate the computation rate. We label the two circuits C_1 and C_2 where $C_1 = \{Q_1, Q_2, Q_3, Q_4\}$, $C_2 = \{Q_2, Q_3, Q_4\}$. The numbers of processes in C_1 and C_2 are 4 and 3 respectively.

From Theorem 2 it follows that the processes are firing periodically after a transient phase, with the computation rate

$$f = \min \left\{ \frac{1}{6}, \frac{(4 - \#\{\text{Discards in } C_1\})}{15}, \frac{(3 - \#\{\text{Discards in } C_2\})}{9} \right\}.$$

We present in Figure 3 an example of the Petri net model we have developed. As conflicts occur, the model of each process is a priority system. The different criteria for process Q_2 are expressed in the different models for process Q_2 (see Figure 4 and Figure 5).

Figure 6 and Figure 7 represent the computing and idle times of the processes. A value discarded is symbolised by a crossed arrow, the abscissa of the vector corresponding to the sending time.

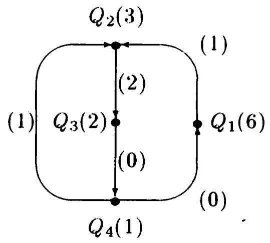


Figure 2. An example of four processes in a strongly connected network. The numbers in brackets are the constant computation or communication times.

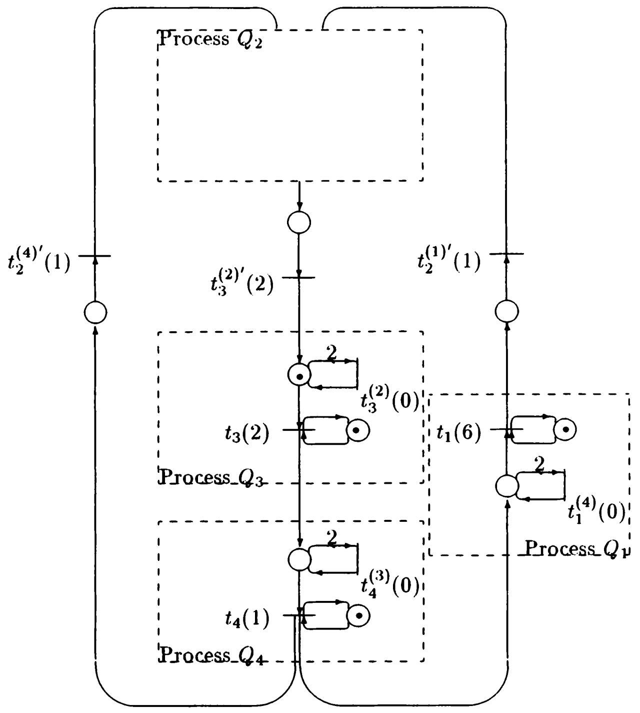


Figure 3. Petri net model of the example network at $\theta = 0$. Transition $t_1^{(4)}$ has priority of firing over t_1 when they are enabled at the same time.

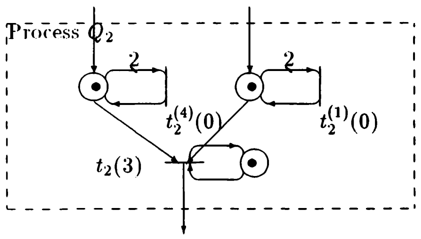


Figure 4. The model of process Q_2 when the criterion is $\mathcal{H}_2 = \{(1,4)\}$. Process Q_2 re-starts only after receiving new values from both process 1 and process 4 – the criterion 'all'.

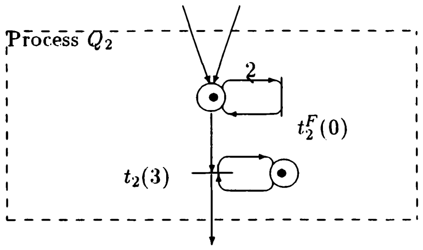


Figure 5. The model of process Q_2 when the criterion is $\mathcal{H}_2 = \{(1),(4),(1,4)\}$. Process Q_2 re-starts when at least one new value is received – the criterion 'one'.

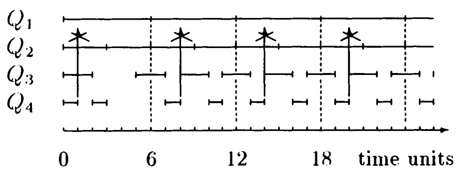


Figure 6. $\mathcal{H}_2 = \{(1),(4),(1,4)\}$. Process Q_2 restarts with at least one new value. The general period is $\pi = 6$ but the computation rates are different so that $K_1 = 1$ and $K_2 = K_3 = K_4 = 2$. $x_4^{(1)}$ is periodically discarded in C_1.

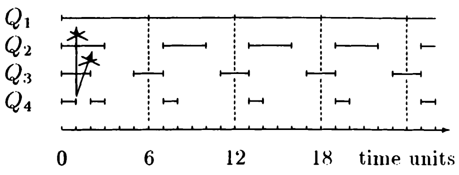


Figure 7. $\mathcal{H}_2 = \{(1,4)\}$, that is criterion 'all'. $x_4^{(1)}$ is discarded once in C_1 and C_2. $n_1 = 4$, $n_2 = 3$ and $f = \min\left\{\frac{1}{6}, \frac{(3-1)}{9}, \frac{(4-1)}{15}\right\} = \frac{1}{6}$. The schedule is 1-periodic with $\pi = 6$.

5 CONCLUSION

We have developed a Petri net model for a class of asynchronous distributed iterative algorithms in which a given processor computes one component of the iterate vector. Each of these processors computes a new iterate as soon as new values of a certain number of the components it requires are received from other processors. Using the Petri net model we have shown that, after a certain initial time has elapsed, the computations all become periodic, and, implicitly, synchronized. For a strongly connected net and when a process requires new values from all the adjacent processors in the dependency graph we have given an exact formula for the period. However, this can only be evaluated once the behaviour of the algorithm during the initial phase is known so we also give an upper bound for the period which can be computed at the outset. Further work is under way to derive an upper bound for the period when more general criteria are used.

6 REFERENCES

Aggarwal, S., Barbara, D. and Meth, K.Z. (1987) Spanner – A tool for the specification, analysis and evaluation of protocols. *IEEE Transactions on Software Engineering*, **13**, 1218–1237.

Alur, R., Itai, A., Kurshan, R.P. and Yannakakis, M. (1995) Timing verifiation by successive approximation. *Information and Computation*, **118**, 142–157.

Baudet, G.M. (1978) Asynchronous methods for multiprocessors. *Journal of the Association for Computing Machinery*, **25**, 226–244.

Bertsekas, D.P. and Tsitsiklis, J.N. (1991) Some Aspects of Parallel and Distributed Iterative Algorithms – A Survey. *Automatica*, **27**, 3–21.

Best, E. and Koutny, M. (1992) Petri net semantics of priority systems. *Theoretical Computer Science*, **96**, 175–215.

Bull, J.M. and Freeman, T.I. (1992) Numerical performance of an asynchronous Jacobi iterations. *Lectures Notes in Computer Science* **634**, 361–366.

Carlier, J., Chretienne, P. and Girault, C. (1985) Modelling scheduling problems with timed Petri nets. *Advances in Petri Nets, APN '84* (ed. G. Rosenberg), *Lecture Notes in Computer Science* **189**, 62–82.

Carlier, J. and Chretienne, P. (1988) Timed Petri nets schedules. *Advances in Petri Nets, APN '88* (ed. G. Rosenberg), *Lecture Notes in Computer Science* **340**, 62–84.

Chretienne, P. (1983) Les reseaux de Petri temporises. These d'Etat, Universite de Paris VI.

Fletcher, L.R. and Santini, M. (1994) Periodicity in an asynchronous algorithm for parallel processing using timed Petri nets. *Proceedings of the HKIWNDCM 94*, Hong Kong.

Fletcher, L.R. and Santini, M. (1995) Periodicity in an asynchronous algorithm for parallel processing using timed Petri nets – Detailed results and proofs. *Technical report, MCS-95-18,* University of Salford.

Murata, T. (1989) Petri Nets: properties, analysis and application. *Proc. IEEE*, **77**, 541–580.

7 BIOGRAPHY

Leslie Fletcher is Head of the Department of Mathematics and Computer Science at the University of Salford. He holds a B. Sc. degree in Mathematics from the University of Manchester and a D. Phil. degree from the University of Oxford. He has published papers in many areas of mathematics and its applications in engineering, finance and medicine. His current research interests include the modelling and control of decentralised systems, from both theoretical and practical standpoints.

Marie Santini is a doctoral student at the University of Salford. She holds a Diplôme d'ingénieur from the Ecole Centrale de Lyon. She is currently researching in the area of parallel processing.

20

Performance indices to characterise concurrent applications: experimenting GSPN evaluation techniques in plant automation

Oliver Botti
ENEL Società per Azioni
Centro Ricerca di Automatica (CRA)
Via Volta 1, Cologno 20093 Milano (ITALY)
e-mail: botti@cra.enel.it
fax. ++2.7224.5525

Lorenzo Capra
Università degli Studi di Milano
Dipartimento di Scienze dell'Informazione
Via Comelico 39 - Milano (ITALY)
e-mail: petrilab@hermes.mc.dsi.unimi.it
fax. ++2.5500.6276

Abstract

This work is part of an investigation aiming at experimenting the use of Generalised Stochastic Petri Nets to model and to evaluate concurrent applications over their target parallel architectures. Former work proposed a modular methodology integrating the modelling and the performance evaluation of both sw applications and hw architectures. Its experimentation over real case studies pointed out the need to deepen some specific steps of model construction and analysis. We here collect the main considerations emerged during the experimentation activity. In particular, we discuss the experienced performance metrics and parameter assignment criteria; then we focus on the definition and use of a set of quantification indices, which have revealed to be suitable to characterise an application in terms of its performance and to support its mapping over a parallel architecture. A portion of a case study taken from ENEL R&D activity in power plant automation is used to show the usefulness, efficacy and expressiveness of the above novelties. Guided by the needs of the industrial final user to which the methodology is oriented, keywords of the whole work have been simplicity of use and flexibility, aiming at increasing practicability and (re)usability in a wide applicative area.

Keywords

GSPN, Process-Box, concurrent applications, performance prediction, performance indices.

1 INTRODUCTION

The strong performance and fault-tolerance requirements imposed by modern automation systems and the parallel and distributed solutions proposed to cope with them point out the central role now played by the design and performance evaluation activities during the system life-cycle. In particular, it reveals to be necessary to integrate performance evaluation since the early design phases. With the aim of allowing a performance oriented parallel system design, a modular methodology based on Petri Nets (PN) was proposed in Botti and De Cindio (1993) integrating modelling and performance evaluation of both sw applications and hw architectures. The interest in PN is due to many reasons: they are a formal model suitable to deal with the several issues of concurrence, whose cognitive effectiveness has shown to favour user's acceptance; experiences and results are known of PN application to support different phases of system development; several computer aided environments are now available to support PN models construction and analysis. The adopted GSPN (Generalised Stochastic PN) class of nets (Marsan (1992)), supported by the GreatSPN tool (Chiola (1987)), allows to integrate formal description, proof of correctness and performance prediction of concurrent and distributed systems. Within this PN based design approach a special role is played by the modelling of a concurrent application oriented to extract its performance characterisation, i.e. its main time related aspects. This has revealed to be useful for obtaining a higher confidence level with time critical issues and for guiding the mapping of the application over the target parallel architecture. Experimenting the proposed approach over real case-studies (e.g. Botti et al. (1995a-b)) pointed out the need to deepen some specific steps of model construction and analysis. In particular, we here consider a few topics which are often neglected in research activities involving stochastic modelling, showing their relevance to correctly obtain and interpret the evaluation results of real-life cases: 1) the definition of the most convenient abstraction level, 2) the definition of proper parameter assignment criteria, 3) the selection of suitable performance metrics useful to characterise a concurrent application and the congruent definition of the corresponding quantification indices. The balance of the paper is as follows. We briefly and informally recall GSPN definitions and the adopted (modular) modelling and evaluation techniques (§2). We deepen motivations and guidelines followed in defining a basic set of performance indices needed to characterise an application, giving their formal definitions and discussing their peculiarities (§3). We experienced efficacy, usability and flexibility of the evaluation technique and of index definitions on a case-study taken from ENEL R&D activity in power plant automation, reporting in §4 a small result selection.

2 MODELLING AND EVALUATION TECHNIQUES: GSPN AND P/R-B

GSPN have been introduced as a tool to allow the integration of formal description and performance evaluation of concurrent systems. While specification and qualitative analysis are supported thanks to the underlying untimed PN class (Place/Transition nets enriched with priorities and inhibitor arcs), quantitative analysis is allowed by the explicit representation of time within GSPN: a random firing delay with negative exponential pdf is associated with some timed transitions, representing the completion time of an activity. The parameter λ of the pdf is the mean firing rate of t (therefore λ^{-1} is the mean firing delay). GSPN provide also immediate transitions, which fire in zero time and with priority over timed ones, to represent logical or time negligible activities. If a marking M enables only timed transitions (in this case M is called tangible) a race firing policy is adopted: for each transition t, an instance of the associated random firing delay is sampled, and a timer is set at this value; while t is enabled, the timer is decreased at a (possibly marking dependent) constant speed. When the timer holds zero, transition t fires. Resampling or an age memory policy can be equivalently adopted with this firing policy, due to the memoryless property of exponential pdf. When a marking M enables at least one immediate transition (in this case M is called vanishing), a preselection is made, based on weights, of one among conflicting immediate transitions with the highest priority. The preselected transitions fire concurrently. Immediate transitions are suitable to represent shared resource acquisition, enhancing a selection on a purely probabilistic, non temporal, basis. As a result of this time representation, a GSPN is isomorphic to a CTMC (Continuous Time Markov Chain), for which well known analytical techniques can be applied to compute the distribution vector over the state space, either in equilibrium conditions (steady state solution), which is guaranteed if the CTMC has one maximal strongly connected component (e.g. if the GSPN is cyclic), or at any arbitrary time instant (transient solution). GSPN have a great advantage w.r.t. CTMC in terms of easiness, evocativeness and immediateness, both in model construction and evaluation. Besides, the availability of a formal model which supports also qualitative analysis (e.g. verification of deadlock freeness, liveness,...) plays an important role. Even though deterministic models should be preferred to specify real time systems, a probabilistic approach, and particularly the markovian one based on the exponential pdf, allows to face with the great complexity of real systems, and it is particularly useful when the goal is, as in our intention, a performance prediction over design solutions.

To enhance the effectiveness and (re)usability of GSPN models, we previously defined the so called Process/Resource-Box (P/R-B) methodology, which guides their construction and use. We first developed, within the Esprit project DEMON, the so called Process-Box (PB) (Hopkins et al. (1992)). A PB consists of a labelled net which describes the control flow of a process, having two sets of places (entry/exit) and two sets of transitions (input/output), called box interfaces, representing the begin/end of the control flow, respectively, the interaction with the environment. Elementary PB may be composed by means of their interfaces to represent more complex structures. In Best and Koutny (1995) the PB approach evolved in a box-algebra based on high level nets allowing a full representation of data. For the purpose of performance evaluation we have integrated the basic PB with GSPN, first introducing a simplified representation of hw components (the available processors) to compare different OCCAM program placements over a target architecture (Botti and De Cindio (1992)), then extending this representation to other hw resources (links, memories, busses,...), giving rise in Botti and De Cindio (1993) to the Resource-Box (RB). Intuitively, a RB of an elementary resource consists of a labelled net whose places represent the resource states, provided with a set of transitions (called service interfaces) representing the basic services offered by the resource. In terms of P/R-B, the mapping of a process over a resource is obtained composing the associated PB and RB. Elementary RB may be composed to produce a compound RB (e.g. the RB of a Transputer node) offering 'macro services' (e.g. assignments, input, output) to the environment (e.g. an OCCAM application). It is worth noting that all the box compositions rely on the same net operators (basically place and transition multiplication), applied to box interfaces. The P/R-B methodology provides PN with modularity and compositionality, enhancing their effectiveness and (re)usability in modelling complex systems. The compositional (step-wise) model construction and the related modular analysis, not only allow to face with model complexity, they also maintain the original system structure within the model, reducing the level of ingenuity requested to the user.

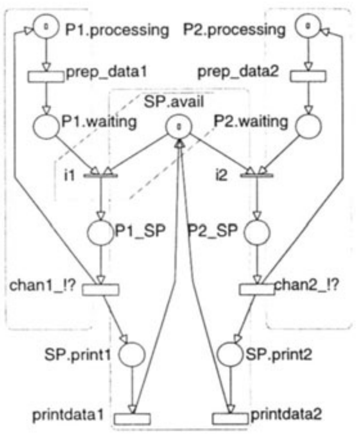
The experimentation of the above methodology emphasised the key role played within the performance analysis by the choices of the parameter assignment criteria and of the abstraction level. These heavily depend on the goals of the analysis, on the system complexity, and on the availability of performance input data (e.g. hw component data sheets, experimental measurements, statistical information...). We sketch in the following the basic choices we have experienced. Firstly, being the application performance characterisation our main goal, we model the sw application in detail, while summarising the hw platform. Secondly, we select the more significant application instructions w.r.t. the resource utilisation and we classify them, according to the carried out activities, in processing instructions and communication instructions. For each class we select a representative instruction, whose real execution time can be given as input data, and which is represented as atomic within the model: the assignment of an integer variable and the transmission of a single byte. Then we define two counter variables, ninstr and nbyte, representing the number of instructions forming an operation burst. We assign to each significant transition a mean firing delay (λ^{-1}) equal to the execution time of the proper basic instruction multiplied by the proper burst length counter. Modifying the counter values we may vary the model granularity, ranging from single basic instructions to bursts. This simple approach allows 1) to refer transition timing to known values characterising the system (the basic execution time includes hw and sw resources), 2) to move the abstraction level, grouping basic instructions into bursts, 3) to make the model parametric w.r.t. burst length. Our model therefore represents a variable workload w.r.t. both the total amount of operations and the processing/communication ratio (ρ = nbyte÷ninstr) of the application. This parametricity is also easily reproducible in the case study (§4), where we compute the values of the proposed performance indices varying the stochastic parameters within proper ranges, then obtaining figures which guide the application performance characterisation and its mapping.

3 A FLEXIBLE SET OF STOCHASTIC PERFORMANCE INDICES

To carry out an appropriate quantitative analysis of concurrent applications one needs to evaluate a set of metrics -execution/response time, communication/processing cost, reactivity, resource utilisation- covering the multiform issues of distributed computing. GSPN modelling technique allows to define several quantification indices with small effort. This is why in GSPN literature translating performance metrics into suitable indices has often been considered trivial. The experience over real case-studies put instead in evidence some important issues, dealt more carefully only in recent works (see Balbo et al. (1992a-b)), that we here deepen:

l the need of verifying that index definitions match the corresponding metrics (congruence);
l the need of providing final users of GSPN models, not necessarily experienced with MC theory, with a set of indices simple to be specified and used, covering the set of basic performance metrics (usability);
l the great convenience to have at disposal indices which are easily adaptable to the features of specific concurrent programming languages (flexibility).

Our effort has been spent in setting up a basic set of indices matching as much as possible the above requirements, customising existing index definitions selected from GSPN bibliography and introducing new definitions. The adoption of a structured evaluation approach based on P/R-B is a significant contribution of the paper. This approach, leading to an uniform representation of sw and hw components, make us confident that the indices here proposed to characterise sw could be used, more generally, to analyse distributed systems. We have experienced index flexibility over two application domains, OCCAM and MML, characterised by synchronous, respectively asynchronous communications. We here restrict our attention to the steady-state GSPN solution, more suitable for performance evaluation, omitting for space reasons any consideration about the possible applications of the transient solution to dependability analysis. We provide also a classification of performance indices which reflects their interpretation: indices yielding an absolute (i.e. dimensioned) quantification (§3.1); indices yielding a relative quantification (expressed in terms of probability) (§3.2); relative indices quantifiable as absolute, which we may call absolutised (§3.3). This classification allows to carry out a quantitative analysis according to different perspectives. To guide the explanation we use a simple example, a cyclic printing spooler that alternatively schedules data coming from two processes (a process represents a concurrent unit that can be allocated on a node of a hw architecture), whose GSPN model, obtained via the compositional PB approach sketched in §2, is depicted in Figure 1 (where box components are highlighted) together with its pseudo-code (in a OCCAM-like notation). Notice that a) the PB do not have entry/exit interfaces due to the cyclic structure of the processes and b) the competition for the shared spooler is modelled by conflicting immediate transitions. Although a high abstraction level is adopted, most of typical issues of distributed computing -concurrence, non determinism, conflicts, synchronisation- are represented. The values of unitary processing/communication times (upt/uct) and of the counters ninstr/nbyte are arbitrarily set.



```
PAR(P1, SPOOL, P2)
where
 P1:                    P2:
 WHILE(TRUE)            WHILE(TRUE)
   SEQ                    SEQ
     prep_data(data1)       prep_data(data2)
     chan1 ! data1          chan2 ! data2

 SPOOL:
 WHILE(TRUE)
   ALT
     chan1 ? buffer
       print(buffer)
     chan2 ? buffer
       print(buffer)

 parameter values:
 uct=upt=ninstr=1;
 nbyte_chan1_!? =10;
 nbyte_chan2_!? =20;
```

Figure 1 GSPN model and pseudo-code of a printing spooler.

Index definitions meet the GreatSPN1.5 syntax: they are algebraic expressions involving P{c} (the probability that a condition c holds, where c is a logical expressions on place marking #p) and E{#p} (the expected number of tokens in place p). Since c corresponds to a set $\{s_i\}$ of states of the underlying MC, $P\{c\} = \Sigma_{s_i \in c}\, \pi_i$, where π_i is the *i*th component of steady state distribution vector π. We use <ec_(t)> as abbreviation of the enabling condition of t: since in our models arcs do not have multiplicity, $\text{<e\_c(t)>} = (\wedge_{p_i \in {}^\bullet t} (\#p_i>0)) \wedge (\wedge_{p_j \in t^h} (\#p_j=0))$, where $^\bullet t$, $t^\bullet$, t^h denote the set of input, output, respectively inhibitor places of t.

3.1 Absolute indices

The absolute indices are based on the throughput of a transition t (thr(t)), that gives the expected number of firings of t at the time unit. If t is timed $\text{thr}(t) = P\{\text{<e\_c(t)>}\}\cdot\lambda_t$, where λ_t is the firing rate. Throughputs of immediate transitions -automatically computed by GreatSPN- have a more complex form which depends on the throughput of timed ones.

Execution/Response time. The mean execution time of a terminating program, whose PB is made cyclic by an auxiliary transition t_0 such that $^\bullet t_0$ and $t_0^\bullet$ are the entry, respectively the exit interface of PB, is quantified by the ADTN (Average Delay in Traversing a Net, Botti and De Cindio (1992)), i.e. by either $\text{thr}(t_0)^{-1} - \lambda_0^{-1}$ or $\text{thr}(t_0)^{-1}$, according to whether t_0 is timed or immediate. In case of reactive programs previous expressions may be computed over suitable transitions, without the need of an auxiliary one, and may be interpreted as a response/completion time.

Absolute costs. Evaluating the amount of communication/processing performed by the processes of an application is crucial to optimise its mapping over an hw architecture. The communication/processing transitions are easily recognisable in the PB structure: the former belong to the I/O interfaces, the latter to the internal structure (see Figure 1). A definition of communication cost is given in Balbo et al. (1992b): denoting with $T_{com(i,j)}$ the set of timed transitions obtained by merging the I/O interfaces of P_i and P_j PB, this cost is defined as A_com_cost1$(i,j) = \sum_{t\in T_{com(i,j)}}$ thr(t), and represents the communication frequency between P_i and P_j. The parameter assignment proposed in §2 allows to define an alternative communication cost, which evaluates the amount of bytes transmitted in the time unit between P_i and P_j:

A_com_cost2$(i,j) = \sum_{t\in T_{com(i,j)}}$ thr(t)·nbyte$_t$ (1)

In section 3.2 we will put in comparison the different versions of communication cost. Denoting with $T_{proc(i)}$ the set of (timed) transitions inside the PB of P_i we can analogously define the processing cost of P_i (representing the mean number of instructions of P_i processed in the time unit) as A_proc_cost$(i) = \sum_{t\in T_{proc(i)}}$ thr(t)·ninstr$_t$.

3.2 Relative indices

A large part of performance indices used in GSPN literature is simply expressed as a probability P{c}. Since P{c} may be seen as the fraction of time unit condition c holds, we call these indices as relative.

Relative costs. We first define the relative cost C(T) of an activity represented by a set T of transitions, as the probability of executing the activity at steady-state, i.e.:

C(T) = P{ $\vee_{t\in T}$ <e_c(t)> } (2)

The relative processing and communication costs are then obtained instantiating T: R_com_cost(i,j) = C($T_{com(i,j)}$); R_proc_cost(i) = C($T_{proc(i)}$). Let us discuss the communication costs so far proposed. A_com_cost1 works if the length of transmitted messages - nbyte$_t$- is constant. For the spooler e.g. (Figure 1) A_com_cost1(P1, SPOOL) = 0,03139ms^{-1}, A_com_cost1(P2, SPOOL) = 0,03116ms^{-1}: the difference is negligible in spite of nbyte$_{chan1\_!?}$ = 10 and nbyte$_{chan2\_!?}$ = 20. This definition i.e. takes into account A) the structural interdependencies between processes, but unfortunately not B) the length of communications: if the interaction frequency (thr(t)) between processes practically does not depend on the stochastic parameter values due to the application symmetry, a variation of nbyte$_t$ is offset by an equivalent variation of P{<e_c(t)>} (recall that λ_t is set to [nbyte$_t$·uct]$^{-1}$). Obviously, A_com_cost2 (1) takes into account both A) and B). This is true also for R_com_cost, based on (2): in fact P{<e_c(t)>} = [thr(t)·nbyte$_t$·uct], and we obtain R_com_cost(P1,SPOOL) = 0,3139, R_com_cost(P2,SPOOL) = 0,6232, that agrees with the length of the two communication instances (which have the same frequency).

The availability of the parametric form (2) allows several other useful costs to be defined, e.g. the communication cost of P_i (as C($T_{com(i)}$), where $T_{com(i)} = \cup_{j\neq i} T_{com(i,j)}$), the processing/communication costs of the whole application (as C($\cup_i T_{proc(i)}$), respectively C($\cup_i T_{com(i)}$)), and the inactivity cost of P_i (as 1-C($T_{proc(i)} \cup T_{com(i)}$)).

Relative reactivity. The reactivity is the most important evaluation metrics for cyclic programs; it estimates the 'readiness' of a server process P_j to react to requests coming from a client P_i. In our models we can identify couples of places <req(i,j), avail(j)> -see Figure 2(a)- that represent the queuing of requests sent by P_i to P_j, respectively the availability of P_j. An intuitive reactivity index, proposed in Balbo et al. (1992b), is the probability of having a request waiting for P_j, i.e. P{#req(i,j)>0 }. This quantification is strictly related to the arrival rate of requests: if it were very low, we should obtain a low probability, improperly meaning a high reactivity. A partial solution is based on net reduction depicted in Figure 2(b), possible (i.e. leading to a stochastically equivalent model) when no immediate transition is in conflict with acquire_j (Chiola et al. (1991)): the reactivity index we propose (3) evaluates the readiness of the server given that some requests *have been* made:

$$R = P\{\#avail(j)>0 \wedge \#req(i,j)>0\} \div P\{\#req(i,j)>0\} \quad (3)$$

Definition (3) works because it considers service rate when some tokens (i.e. requests) are in req(*i,j*) place (notice that for the GSPN in Figure 2(a) expression (3) holds always zero, since the simultaneous marking of req(*i,j*) and avail(*j*) is vanishing). Unfortunately, we cannot do the previous reduction when P_j is a shared resource (e.g. in Figure 1, where P_j corresponds to SPOOL). We overcome this restriction in §3.3, giving an absolute reactivity definition.

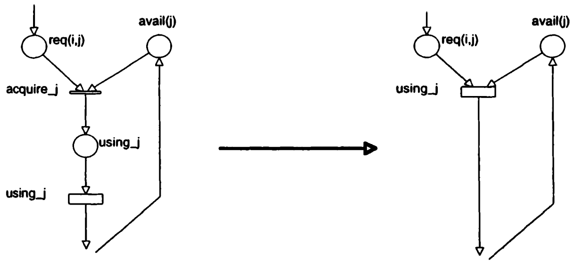


Figure 2 (a) Interaction between a server and a client; (b) its structural reduction.

Resource utilisation. Thanks to P/R-B approach it results very easy to define and compute an important performance index as resource utilisation, where resource may equivalently be hw or sw. If, as in our models, a place res_i is associated with *i*th resource (#res_i = 0: resource busy; #res_i = 1: resource idle), the *i*th resource utilisation is given by P{#res_i=0}: the spooler utilisation (Figure 1), e.g. is P{#SP.avail=0}. More generally, if place res_i represents a set of n resources of kind *i*, then the mean utilisation (MU) of a resource (of kind *i*) is defined as:

$$MU = (n-E\{\#res\_i\})\div n \quad (4)$$

3.3 Absolutised indices

A relative quantification of performance metrics is surely suitable for characterising a concurrent application. Nevertheless, an absolute interpretation of such relative metrics often reveals to be profitable, and sometimes necessary.

Absolutised costs. It is very simple to give an absolute version of the parametric cost (2) in case of terminating programs. We define the absolutised cost $C_a(T)$ as:

$$C_a(T) = C(T)\cdot ADTN \quad (5)$$

$C_a(T)$ evaluates the fraction of mean execution time spent by a program in the activity represented by T. As usual, we obtain the processing/communication costs by instantiating T. We have selected from the bibliography on Markovian models a more transparent and general way of quantifying the absolutised costs. Let c be a condition holding on a set of markings of an ergodic GSPN A, **Q** the infinitesimal generator of the underlying MC, $\underline{c}$ the (complementary) set of states where c does not hold. Using simple linear algebra it is possible to compute the expected time c holds, denoted in Ciardo and Trivedi (1991) (where we demand for more details) as $\mu_A(c)^{-1}$. It is: $\mu_A(c)^{-1} = (\Sigma_{s_i \in c}\, \pi_i) \div (\Sigma_{s_i \in c,\, s_j \in \underline{c}}\, \pi_j \cdot \mathbf{Q}_{j,i})$. A congruent definition of absolutised cost of an activity (represented by T) will therefore be:

$$C'_a(T) = \mu_A(<ec\_(\ T\ )>)^{-1} \qquad (6)$$

The absolutised communication cost of SPOOL process -Figure 1- e.g. is $\mu_A(<ec\_(\{chan1\_!?,\ chan2\_!?\}\ )>)^{-1}$, and represents the fraction of cycle time spent by the spooler in communication. Since the computation of $\mu_A(c)^{-1}$ requires an explicit knowledge of the underlying MC, it becomes practically unmanageable for GSPN of mean dimensions. We aim at enriching GreatSPN to automatically compute $\mu_A(c)^{-1}$from a net level specification of c.

Absolutised reactivity. To give an absolute reactivity we have to compute the average delay in traversing a subnet (here called LADTN, i.e. Local ADTN), defined in Marsan (1989) as extension of LITTLE formula to GSPN. Let SN = (P', T', ...) be a subnet of a GSPN A = (P, T, ...) and P_{in}, T_{out} be the set of input places, respectively of output transitions of SN (the interfaces: $P_{in} = \{p \in P' : {}^\bullet p \cap (T - T') \neq \varnothing\}$; $T_{out} = \{t \in T' : t^\bullet \cap P' = \varnothing \wedge t^\bullet \cap P \neq \varnothing\}$); $LADTN_{SN}$ is the ratio between the expected number of tokens traversing SN and the mean output token flow:

$$LADTN_{SN} = (\Sigma_{p \in P'}\, E\{\#p\}) \div (\Sigma_{t \in T_{out}}\, thr(t)) \qquad (7)$$

The LADTN has a meaning if the paths in which we are interested can be located and a fixed ratio exists between the number of tokens entering SN and the firings of output transitions. Checking these conditions is generally difficult due to the complexity of real system models. This is why formula (7) is practically never used. The only exception known to the authors is the decomposition methodology of large GSPN, based on LADTN, given in Murray and Li (1991), where subnets must be Marked Graphs and connected with the context only through their interfaces ($\forall t \in T' : {}^\bullet t \cap (P - P') = \varnothing$). We are interested in computing LADTN of subnets not necessarily limited to the previous restrictions. The location of interesting paths is made easier by the box structure, where box places represent states of a process. To formally check the token flow conservation, we put some structural restrictions on SN (orthogonal w.r.t. Murky and Lie (1991)), not limiting the usability of LADTN as reactivity index:

$$\forall t \in T - T' : ({}^\bullet t \cap P' = \varnothing) \wedge (t^\bullet \cap P' - P_{in} = \varnothing) \quad (8) \qquad \forall t \in T_{out} : t^\bullet \cap P' = \varnothing \quad (9)$$

$$(\forall t \in T' - T_{out} : |t^\bullet \cap P'| = |{}^\bullet t \cap P'| = 1 \wedge (t^\bullet \cap {}^\bullet t \cap P') = \varnothing\ ) \wedge (\forall t \in T_{out} : |{}^\bullet t \cap P'| = 1) \qquad (10)$$

Above conditions assure that (8) tokens traversing SN are not diverged to the outside and from the outside no token is put into SN but into input places, (9) the output transitions of SN let only tokens leave, (10) locally the firing of every transition of SN takes one token from a place of SN and put it into a different one. Moreover, to make the LADTN interpretation really clear, we assume that $|P_{in}| = 1$. If in so formed SN there is no cyclic chain $p_{k_1} t_{k_1} \ldots p_{k_1}$ ($t_{k_i} \in T'$, $p_{k_i} \in {}^\bullet t_{k_i} \cap P'$) which does not belong to any traversing chain $p_{in} \ldots t_{out}$, ($p_{in} \in P_{in}$, $t_{out} \in T_{out}$), every token entering SN leaves it via T_{out}, but for external interactions. Let us consider again the Figure 1. The dashed subnet SN_1 we are interested in is composed by the input place P1.waiting and the output transition i1, and meets conditions (9-11); $LADTN_{SN_1}$ correctly evaluates the spooler reactivity towards P1, representing the waiting time of a request before being scheduled, once it has arrived in place P1.waiting. For the assigned transition rates we obtain $LADTN_{SN_1} = E\{\#P1.waiting\} \div thr(i1) = 2.085$(ms).

By means of LADTN it is possible to give an absolute evaluation of reactivity (and similar metrics) even when there is competition for shared resources. Structureness of box approach and checks on subnet structure can help inexpert users towards a correct LADTN specification, otherwise enough complex. We are actually trying to adapt formula (7) to SN not conserving the token flow (for which (8) does not hold), that should furthermore enhance the LADTN usability, for the computation of metrics which are not related to the reactivity.

4 EXPERIMENTING EVALUATION TECHNIQUES ON MML

The MML language (Multi Micro Language) and the related MME environment (TxT (1993)) have been developed during the early 80s with the financial contribution of ENEL-CRA, as an integrated programming environment for the development of sw applications in the field of real time distributed plant automation systems. The language is based on the Pascal ANSI in its sequential part, provided with primitives to express communication and synchronisation between processes. MML programs are a fixed set of concurrent processes, called sequences. Each sequence may contain local, remote and interrupt procedures synchronising both on Boolean conditions, by RETEST statements, and on queues, by WAIT and SIGNAL statements. Sequences communicate between each other via Remote Procedure Call (RPC). We selected a testing MML program, including the essential features of the language, and a target two-processor architecture to experiment the proposed evaluation technique. Our goal is here mainly to experience usability and flexibility of the proposed indices w.r.t. a specific framework. The program is composed by six sequences: a Driver, a Monitor, a Dispatcher and three Receivers. The Driver accepts characters coming from the keyboard (via an interrupt procedure), and makes them available to the Monitor, which collects the characters into arrays of a prefixed length via RPC to the Driver. Messages are transferred from the Monitor to the Dispatcher (via RPC), and then are distributed to one of the Receivers (via RPC to Newdata procedure) for the processing; the Main and Newdata procedures of Receivers synchronise on a semaphored queue. Concurrence is mainly present among the processing activities of the Receivers. The target hw (still significant within the ENEL plant automation, in spite of its old processors) is composed by two processing units, based on a DEC LSI11/73 (PU1) and, respectively, a Motorola MC68010 (PU2) microprocessor, provided with a local memory and an internal bus. These are connected by a (9600 baud) serial link.

For space reasons we only sketch the construction steps of Receiver[1] GSPN model (Figure 3). Dashed lines point out the PB of Newdata and Main procedures (letters A and B) and the RB associated with the queue pseudo variable (C) and the mutex resource (D) (a mechanism implicit in MML to guarantee the proper execution of each sequence procedure). Except for box D, each component model is obtained from the program code, starting from the PB of elementary language constructs, through the sequential composition rules. According to the approach adopted in Balbo et al. (1992a), data representation is limited to the control variables (the queue pseudo-variable). As concerns the control flow, we explicitly model the wait/signal primitives, as they represent local synchronisation, and the begin/end instructions, as they represent synchronisation via RPC with the Dispatcher sequence. The remaining instructions are collapsed into single transitions (with post fix t_i). The whole model in Figure 3 is obtained applying the parallel composition rule to A, B, C, D boxes. Table 1 reports the meaning of timed transition of Figure 3 and their firing rates. Immediate transitions represent, as usual, the competition for shared resources (e.g. the sequence mutual exclusion). The planned performance analysis aims at characterising the application by the comparison of three different configurations over the target architecture. For that, as sketched in §2, we have selected the transitions more significant w.r.t. resource utilisation, recognisable in Table 1 from the parametric firing rates. Default firing rates have been assigned to remaining transitions. The parametricity w.r.t. λ (the interrupt frequency), ninstr, nbyte makes it possible to study a program class, enhancing the representativeness of the modelled workload.

```
Receiver sequence
queue semaphore;
deferred ind: integer;
var     bufstring: str;

procedure newdata (word: str);
begin
   bufstring := word;
   signal (semaphore);
end;

begin /*main */
  repeat
   if (ind = 1) then
   .../*processing by rec. 1 */
   if (ind = 2) then
   .../*processing by rec. 2*/
   if (ind = 3) then
   .../*processing by rec. 3 */
   wait (semaphore);
   .../*processing of message*/
  until (false)
end.

/* in the main of Dispatcher sequence */
   .../*collecting data; selecting a Rec. */
   call receiver[ind].newdata(word);
   ...
```

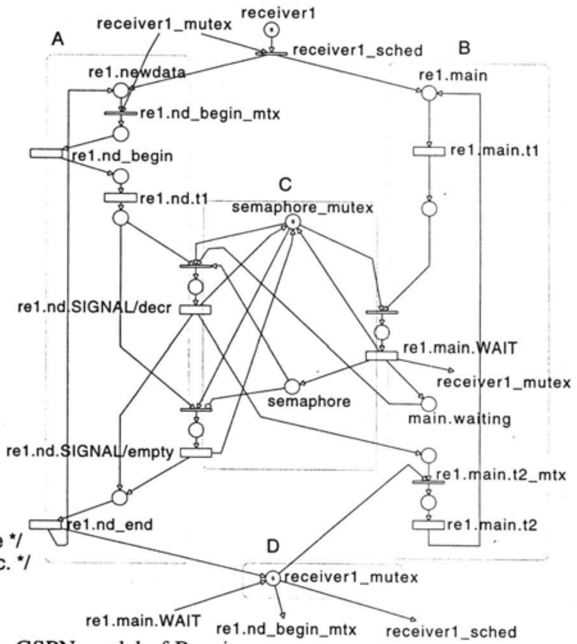


Figure 3 Code and corresponding GSPN model of Receiver sequence.

Table 1 Description of Figure 3

Name	Meaning	Firing Rate (μs^{-1})
re1.nd_begin	synchronisation Rec.-Disp. by RPC	$(c_0+uct\cdot nbyte+c_1)^{-1}$
re1.nd.t1	generic processing by newdata	1
re1.nd.SIGNAL/decr	effective signal on semaphore	1
re1.nd.SIGNAL/empty	signal on empty semaphore	1
re1.nd_end	end of remote procedure newdata	1
re1.main.t1	generic processing by main	1
re1.main.WAIT	queuing of main on semaphore	1
re1.main.t2	processing of a received message	$(upt\cdot ninstr)^{-1}$

The considered configurations are: all the sequences allocated on PU1 (C1); the Driver, Receiver[1], Receiver[2] allocated on PU1, the remaining sequences on PU2 (C2); the Monitor and all the Receivers on PU1, the remaining sequences on PU2 (C3). The model of a configuration is obtained composing the PB of the application together with the simplified RB of hw resources, exactly as done with the Main and Newdata PB and with the mutex RB (Figure 3). Every significant transition is properly refined on the basis of employed resources (see Botti and Capra (1996)). We report only a small selection of numerical results obtained with GreatSPN1.5, focusing on the relative Processing Cost of Receiver[1] (PCR), which is defined by $C(T_{proc(Receiver[1])})$ (2), replacing the process concept by that of sequence (PCR = P{<e_c(re1.nd.t1)> ∨ <e_c(re1.main.t1)> ∨ <e_c(re1.main.t2)>}). PCR evaluates the whole system performance: the more it is high, the more the system accomplishes useful work. Figure 4 represents PCR in C1, C2 and C3 as a function of ninstr (in logarithmic scale), for a fixed λ and for several ρ (nbyte÷ninstr) values, which characterise the workload as CPU-bound (low ρ values) or I/O-bound (high ρ values). Figure 4 points out some expected common behaviours, useful for a first validation of the model: PCR grows with ninstr and is higher for CPU-bounded than for I/O-bounded workloads. Instead single curves follow a quite different trend. For very high interrupt frequencies ($\lambda = 10^{-2}\mu s^{-1}$, Figure 4(a)) the processing performed in C1 is much more than in C2 and C3 (the distributed configurations): C1 is undoubtedly the 'best' configuration in this case, that means for high interrupt frequencies the serial link bottleneck prevents the parallelism among sequences from being exploited. Instead if we consider an intermediate frequency ($\lambda = 10^{-4}\mu s^{-1}$, Figure 4(b)) we note the existence of a threshold workload over which C2 and C3 produce more useful work than C1 (the frequency $\lambda = 10^{-4}\mu s^{-1}$ is particularly meaningful since it is closely related to input data rate of high-voltage electricity substations). Notice that, in spite of λ grows of several magnitude orders when passing from Figure 4(b) to Figure 4(a), the maximum PCR values remain nearly the same (about 0,31), meaning that there is a threshold interrupt frequency for the considered sw architecture over which the system performances deteriorate. Therefore not only the target hw, but also the sw architecture, due to the strong interdependencies among the sequences, cannot efficiently manage high interrupt arrival rates. This performance characterisation is confirmed in Botti and Capra (1996), where other interesting metrics are evaluated using the definitions given in §3, e.g. the communication costs of the sequences and the reactivity to an interrupt request.

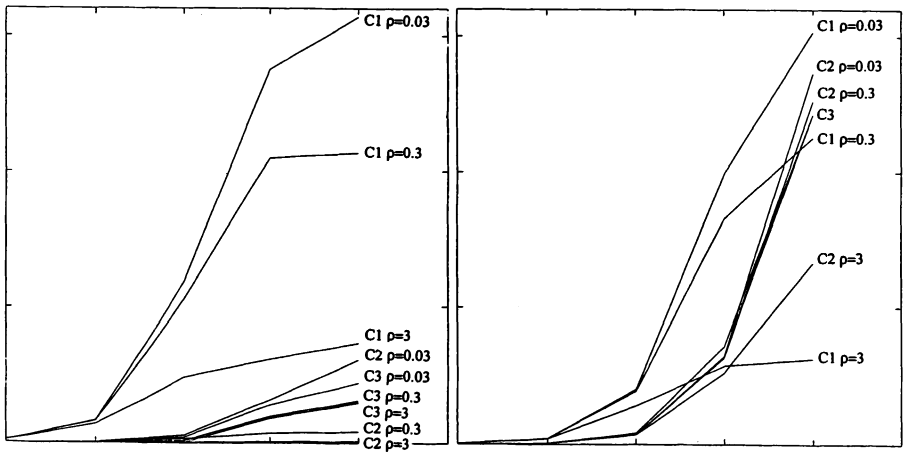


Figure 4 PCR (a) $\lambda = 10^{-2}\mu s^{-1}$ (b) $\lambda = 10^{-4}\mu s^{-1}$ (10 -log x- 10^6; 0 -y- 0,32)

5 CONCLUSIONS AND FURTHER WORK

This paper reports the application of a modular GSPN based modelling and evaluation technique in the field of plant automation, guided by industrial practicability and usability. A set of performance indices is defined to characterise an application and it is experimented on a case-study. Interesting evolutions follow the line of integrating the exponential probabilistic evaluation with non exponential or deterministic ones, adopting a mixed model which allows to suitably describe timing constraints of critical components. Since the solution of such mixed models has often shown to be impracticable, an integrated approach will benefit of the proposed P/R-B modularity and compositionality to separately analyse probabilistic and deterministic submodels, then reusing the obtained partial results in a simplified global model (e.g. on the line of Botti and De Cindio (1993)).

6 REFERENCES

Balbo, G. Donatelli, S. and Franceschinis, G. (1992a) Understanding Parallel Programs Behaviour through Petri Net Models, Special Issue of *Journal of Parallel and Distributed Computing on PN Modelling of Parallel Computers.*

Balbo, G. Donatelli, S. Franceschinis, G. et al. (1992b) On Parallel Programs Characterisation, in Proc. of Conf. *Performance Evaluation 1992.*

Best, E. and Koutny, M. (1995) A Refined View of Box Algebra, in Proc. of the 16th Int. Conf. *Application and Theory of PN,* Torino.

Botti, O. and Capra, L. (1996) A GSPN Based Methodology for the Evaluation of Concurrent Applications Experienced in Plant Automation, submitted to the *Euromicro Journal of Systems Architecture.*

Botti, O. and De Cindio, F. (1992) Comparison of Occam Program Placements by a Generalised Stochastic Petri Net Model, in: Proc. of the Conf. *Transputer '92: Advanced Research and Industrial Applications,* Besancon, France.

Botti, O. and De Cindio, F. (1993) Process and Resource Boxes: an Integrated PN Performance Model for Applications and Architectures, in IEEE Proc. of the Int. Conf. *Systems, Man and Cybernetics,* Le Toquet, France.

Botti, O. Donatelli, S. and Franceschinis, G. (1995a) SWN Models of Parallel Architectures: an Application in the Field of Plant Automation Systems, in Proc. of 16th Int. Conf. *Application and Theory of PN - Case-Studies section,* Torino.

Botti, O. Donatelli, S. and Franceschinis, G. (1995b) Assessing the Performance of Multiprocessor Architectures through SWN Models Simulation: a Case-Study in the Field of Plant Automation Systems, *submitted for publication.*

Chiola, G. (1987) A Graphical Petri Net Tool for Performance Analysis, in Proc. of the 3th Int. Workshop on *Modelling Techniques and Performance Evaluation,* AFCET, Paris.

Chiola, G. Donatelli, S. and Franceschinis, G. (1991) GSPN versus SPN: what is the actual role of immediate transitions?, in IEEE Proc. of the 4th Int. Workshop on *Petri Nets and Performance Models,* Melbourne, Australia.

Ciardo, G. and Trivedi, K.S. (1991) A Decomposition Approach for Stochastic Petri Nets Models, In IEEE Proc. of the 4th Int. Workshop on *Petri Nets and Performance Models,* Melbourne, Australia.

Hopkins, R. Hall, J. and Botti, O. (1992) A Basic Net Algebra for Program Semantics and its Applications to Occam, in G. Rozenberg (ed.), *Advances in PN 92,* LNCS n.609, Springer-Verlag.

Marsan, M.A. (1989) Stochastic Petri Nets: an Elementary Introduction, in G. Rozenberg (ed.), *Advances in PN 89,* LNCS Vol.424, Springer Verlag.

Marsan, M.A. (1992) Generalised Stochastic Petri Nets: Definition at the Net Level, in IEEE *Transactions on Software Engineering.*

Murray, C. and LI, Y. (1991) Performance Petri Net Analysis of Communications Protocol Software by Delay Equivalent Aggregation, In IEEE Proc. of the 4th Int. Workshop on *Petri Nets and Performance Models,* Melbourne, Australia.

TxT (1993) *MML Multi Micro Language: Language Reference Manual,* TxT S.p.A., Cise S.p.A., Milano.

21
Reverse profiling

F.W. Howell
University of Edinburgh
Department of Computer Science, J.C.M.B, The King's Buildings,
Mayfield Road, Edinburgh, EH9 3JZ, Scotland.
Telephone: +44 131 650 5141. email: `fwh@dcs.ed.ac.uk`

Abstract
This paper addresses the problem of designing parallel message passing programs with a reasonable idea of how well they will actually perform before they are run.

Models with very few parameters (e.g. LogP, PRAM) sacrifice accuracy to simplify design. By contrast, simulation techniques provide a good degree of accuracy by incorporating sophisticated architectural models, but present a "black box" to the user. This paper suggests a compromise between the two extremes, using an automatically generated model with a large number of parameters (a separate equation for each MPI function) which is presented to the user rather than being hidden within a black box. The profiling interface of MPI may be used "in reverse" to insert (rather than measure) expected timings from the model.

Keywords
MPI, profiling, performance prediction

1 INTRODUCTION

Programming parallel machines is somewhat of a black art as it is hard to know how well a program will run on a machine before actually running it.

The ideal model for designing parallel programs would be both simple to use and accurate in its predictions. However such a model doesn't yet exist, the simple models which are usable do not predict what actually happens reliably and the models which are fairly accurate (such as the simulation techniques) are both too cumbersome for general use and also present an opaque "black box" view of an architecture whose mysterious inner workings are not exposed. This leads to a development approach similar to the post mortem profiling technique used on actual machines.

The real challenge is to develop an approach which yields useful design information without requiring too much effort on the part of the programmer; if the method is too involved and complex then the programmer won't use it and will revert to post mortem tuning.

The technique of "reverse profiling" addresses some of these problems. There are two strands to the approach:

- The model is automatically generated by running an "MPI characterisation" routine on an architecture, rather than being crafted from in-depth knowledge of the architecture. The model is made available to the programmer for constructing quick pencil/paper analyses of performance.
- Since performing these calculations becomes tedious, especially when evaluating performance on a range of machines and problem sizes, a method is included for automatically computing these delays using the profiling interface of MPI. Rather than use profiling to *extract* timing data from a run of a program, "reverse profiling" *inserts* estimated times.

The performance model consists of separate equations for each MPI function giving the average, minimum and maximum times for a given number of processors and message size. These equations are generated automatically by an MPI program which times each MPI function with a range of message and group sizes, then fits an appropriate equation to the data. Running this on an architecture produces a LaTeXdocument with the equations for each function and graphs of the timing data used to generate the equations. This "datasheet" may be used by the programmer for quick estimates of the time an MPI function will take. A summary file is also produced for the reverse profiling.

The equations given in the model may be used for analytical performance predictions of a program, possibly in conjunction with a spreadsheet or graphing package to experiment with alternative designs at an early stage.

Alternatively the evaluations may be done by the computer using reverse profiling. This involves linking in an extra library, in exactly the same way as a normal profiling interface is linked. The reverse profiling library intercepts each call to an MPI function in the program, uses the appropriate equation to estimate the time the function would take and generates a trace file in a similar manner to a standard profiler. It then calls the normal MPI function to actually perform the communication.

The next section describes related techniques for performance prediction; section 3 describes the routines for generating the model of MPI performance and section 4 details reverse profiling. This is followed by an example and conclusion.

2 OTHER TECHNIQUES

Many approaches have been suggested to tackle the problem of performance prediction; the two ends of the spectrum are simple models like LogP (Culler, 1993) and detailed simulation (Brewer, 1993). Foster (1994) provides an interesting description of parallel design techniques. Driscoll (1995) uses an approach based on an extension of Amdahl's law to look at the performance of a program in terms of equations describing the sequential and parallel sections, a higher level view of performance prediction than the approach of this paper.

Getting closer to the source code level, Sarukkai (1994) addresses the problem of scalability analysis, using the SAGE/SIGMA toolkit to derive a program graph which is analysed to produce a complexity model. Wabnig (1995) also represents the program by a directed graph and the hardware by a processor graph, noting that these graphs get very large for real programs.

LAPSE (Dickens 1993) uses a parallel simulation technique for performance predictions of message passing programs on the Intel Paragon. It uses a simple delay model for point to point communications and provides its own versions of the collective calls written in terms of these.

Reverse profiling is intended as a practical quick approach for the many programmers relying on post mortem techniques at present. It scores over other approaches in providing models directly based on the parallel primitives the programmer sees and in being as straightforward to use as standard profiling. It is not a revolutionary approach; rather a step towards the ideal of pre-natal design rather than post-mortem analysis of parallel programs.

3 GENERATING THE MODEL

It would be useful if performance models for MPI were supplied along with the libraries, but this is not the case, so they need to be generated. A model for point to point communication is not sufficient as much use is made of collective communication calls in MPI, such as `MPI_Bcast`, `MPI_Alltoall`, `MPI_Reduce`, `MPI_Barrier` etc. These all have different performance characteristics which are not adequately described by simple point to point models such as LogP. Parallel benchmarks tend to be directed towards comparing machines rather than providing design data for programmers.

Nupairoj (1995) describes an approach to benchmarking the MPI collective communications which attempts to work out how the structure of the underlying implementation of the collective MPI functions in order to derive reasonable performance models. In contrast the technique described below simply provides equations to *describe* the delays seen by a programmer calling each MPI function. A characterisation run only needs to be performed once for each architecture of interest to generate the required model.

3.1 Measuring performance of MPI building blocks

Characterising the performance of the MPI functions is straightforward in principle; measure the time to complete N calls and take the average. The parameters of interest are the number of processors and the size of the messages.

To time an operation (e.g. `MPI_Bcast()`), a short function is written:-

```
void time_Bcast(int numelems, double &time)
{
  int *buffer = new int[numelems];
  MPI_Barrier( comm );
  double e1 = MPI_Wtime();

  MPI_Bcast( buffer, numelems, MPI_INT, 0, comm );

  time = MPI_Wtime() - e1;

  time = getmax( time );
  delete buffer;
}
```

The `MPI_Wtime()` function is used to time the operation. The processes are synchronised beforehand using an `MPI_Barrier`. This is not perfect, as some processes may return from the barrier before others, so an alternative synchronisation technique has also been used which first determines the clock skew between different processes' `MPI_Wtime()` values, then busy waits until the timer reaches an agreed value. This provides synchronisation to a resolution of the short time required to read the timer, but just using `MPI_Barrier` is more convenient in practice.

The time is measured from this synchronisation point until the last process has returned. The `getmax()` function uses an `MPI_Reduce` across all processes to determine this maximum delay.

The parameters are the size of the message and the number of processes in the current communication group `comm`. These are varied across the range of values of interest on the machine, and each timing is repeated to produce a 3D set of measured times of the operation on the machine.

A surface is then fitted to this data using a least squares technique. It is not known beforehand what form the equation should take. There may be a constant start up cost with a linear data dependent factor for the message to be transferred across the network; or a data dependent startup (corresponding to an initial copy of the message into an internal buffer) with a data independent transfer cost (in a shared memory machine); the time may grow linearly with the number of processors, or with the logarithm of the number of processors for tree based algorithms; there may well be a network contention factor which predominates with large messages. The list of possible factors is endless and varies from machine to machine and from MPI function to MPI function.

Determining all the physical machine and algorithm parameters is not the aim of this approach. The aim is a descriptive equation which is simple enough to use and which provides confidence intervals to indicate the goodness of the fit. No claim is made that the parameters correspond directly to anything in real life; the only claim is that they fit the measured data to a given degree of accuracy.

In order to obtain this elusive compromise between a simple equation and an accurate fit, a brute force approach is taken performing a range of different curve fits and selecting and the best. The equations for the time of an operation in terms of the number of processes in the group p and the message size d take the form of a constant factor, a "startup parameter" dependent on the number of processors, and a "data dependent" factor dependent on the message size and the number of processors:-

$$t(p,d) = \text{c\_coeff} + \text{s\_coeff} * \textbf{startupfn(p)} + \text{d\_coeff} * \textbf{datafn(p,d)}$$

$$\textbf{startupfn(p)} = \text{one of} \begin{cases} p \\ \log(p) \\ p^2 \end{cases}$$

$$\textbf{datafn(p,d)} = \text{one of} \begin{cases} d \\ pd \\ \log(p)d \\ p^2 d \end{cases}$$

Thus a total of 12 curve fits are performed using every combination of the startup and data functions. These functions were chosen as they provide reasonable fits for all cases thus far encountered. It was originally hoped to provide an adequate fit using one or two coefficients but this wasn't sufficient for the collective calls.

A fit is performed to determine the three coefficients using all combinations of the two functions and the one with the minimum chi-squared value is selected. Estimates of the standard error of each coefficient are also produced. These yield equations giving the maximum and minimum expected times. This should only be used as a rough guide, as there is no guarantee (or even likelihood) that the measured data conforms to a normal distribution. However, it is useful to have at least some indication of expected confidence intervals.

An example equation for `MPI_Allreduce` is:-

$$T_{allreduce}(\mu s) = \begin{cases} (50 \pm 30) + (200 \pm 10) \times log(p) + (4 \pm 1) \times d & \text{if } d <= 32 \\ (300 \pm 30) + (20 \pm 2) \times p + (0.9 \pm 0.03) \times log(p) \times d & \text{if } d > 32 \end{cases}$$

Separate equations are given for "small" and "large" messages as the shape of the fit often differs.

3.2 Output formats

The model is intended to be available for programmers to have an idea of the delay imposed by each MPI function. Because of this, one of the output formats is an automatically generated LaTeXdocument listing the equations and giving graphs of both the raw data and the fitted surfaces. Figure 1 gives an example page from a datasheet. The other output format is a summary file for computer based tools (such as the reverse profiler) to read.

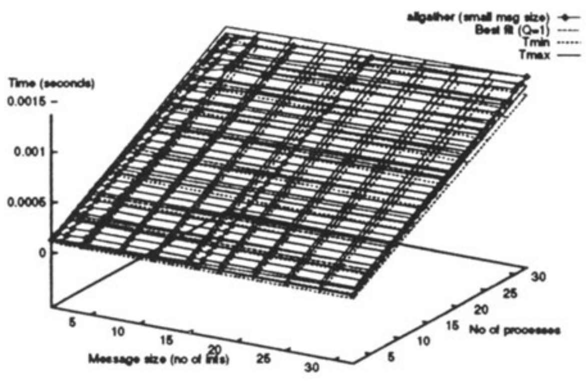
4 REVERSE PROFILING

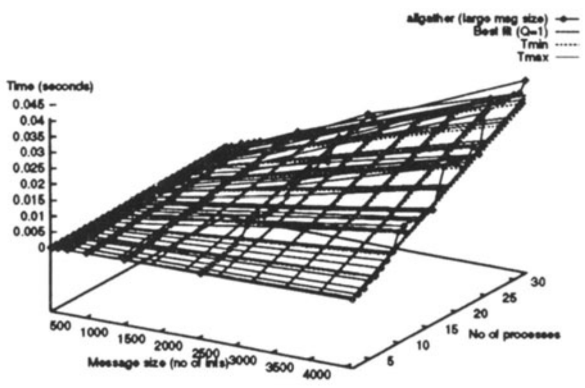
Reverse profiling is a technique which applies the MPI performance model for an architecture to a user's program to generate an estimate of the run time on that architecture. It uses the MPI profiling interface to intercept the user's calls to MPI functions and calculate the expected delay before returning control to the MPI routine to do the actual work.

Each process keeps track of its own simulation time and updates it whenever an MPI function is called. This means a normal trace file can be generated. A model of any machine may be used, and any MPI implementation can be used as the development environment. For example, workstation implementation of MPI may be used with a Cray T3D model to generate predictions of performance on the parallel machine.

Because it does not involve full simulation, it can't be applied to non-deterministic routines, for example those employing dynamic load balancing. However, the performance model will provide the key design data for such routines (such as the minimum and maximum message times). For non-deterministic programs the method must be combined with pencil and paper calculations, or with times measured from the target machine. Note that non-deterministic programs are likely to strain simulators and profilers too, since a minor miscalculation of delay may affect the outcome. A large proportion of useful parallel

allgather





$$T_{allgather}(\mu s) = \begin{cases} (50 \pm 20) + (40 \pm 1) \times nprocs + (1 \pm 0.9) \times ndata & \text{if } ndata <= 32 \\ (4 \pm 20) + (40 \pm 3) \times nprocs + (0.3 \pm 0.009) \times nprocs \times ndata & \text{if } ndata > 32 \end{cases}$$

Figure 1 A page from an automatically generated MPI data sheet.

programs are deterministic. Reverse profiling is a simple usable technique aimed at the majority of programs.

4.1 Results generated using reverse profiling

Running a reverse profiled MPI program produces a trace file which may be displayed as a timing diagram. Repeated runs may be used to produce graphs showing how performance varies with the problem size and number of processors in the machine. The machine model is supplied at run time as an environment variable pointing to a file produced by the MPI characterisation routines.

4.2 The technique in detail

MPI (MPI Forum, 1995) provides a simple profiling interface; all the `MPI_` functions are also accessible with the prefix `PMPI_`. Profiling (or reverse profiling) code may be added by writing substitute `MPI_` functions which perform the necessary (reverse) profiling task and call the `PMPI_` function to do the actual work. The linker ensures that the appropriate functions are called. The compilation commands to compile a normal MPI program, to compile with a profiler and to compile with the reverse profiler are:-

```
cc prog.c -lmpi
cc prog.c -lprof -lpmpi -lmpi
cc prog.c -lrevprof -lpmpi -lmpi
```

Each process has a `double the_time` variable to store its current simulation time. The profiled versions of the MPI functions update `the_time` according to the performance equation for that function and write lines to the trace file.

For point-to-point communications the receiver needs to know the time the sender started sending the message in order to work out when it should arrive. The minimum delay at the receiving end occurs when the message has been posted by the sender well in advance and the message has only to be copied from a system buffer. If the `send` starts at the same time as the `recv`, there will receiver will suffer an additional wait time for the message to arrive. This will be worse if the sender starts after the receive does.

For collective operations involving synchronisation (i.e. the majority of them), each process must know the start time of every other. Thus a point-point reverse profile function looks like:

```
int MPI_Send( data, dest, ...)
{
  // Send the_time to the destination
  PMPI_Send(the_time, dest, ...);
  the_time += /* computed delay for the message */;

  // Perform the actual send
  PMPI_Send( data, dest, ... );
}

int MPI_Recv( ... )
```

```
{
  // Recv the sender's start time
  // Compute the recv delay the_time
  // function of ( the_time, sender_start, message size )
}
```

and a collective operation:–

```
int MPI_Barrier()
{
  // MPI_Allgather to get each process's the_time
  // Set local the_time to the latest of all the_times
  // Plus the computed delay for the barrier.
}
```

This works as long as two conditions are met:

1. MPI_Recv is not allowed wildcarded receives. This is because there are two receives (one for the sender time, one for the actual data) which couldn't be guaranteed to come from the same source. This problem is related to the non-determinism issue raised earlier. A solution would be to tag the timestamp onto the main body of the message, or to do a wildcarded receive for the first message, work out where it came from, and do a receive from there.
2. Collective operations imply synchronisation.

At present a trace file is generated which may be displayed with the HASE timing diagram tool (Howell, 1994). Additional tracing (e.g. source code line numbers) could be added if necessary. Each process generates a separate trace file (`p<rank>.trace`), and repeated runs may be combined to produce scalability graphs.

4.3 Estimating the computation delays

The reverse profiling technique has accounted for the communication costs quite happily, but the times for user code have not been accounted for. Even without considering compute times, useful results may be obtained since the amount of time spent in idle "wait" states can be measured from the timing diagram and the communications structure of the code is clearly visible. None of the techniques thus far encountered by the author for this purpose are entirely satisfactory. In practice a combination of the following techniques for estimating computation time are used, with option 2 yielding the preferred tradeoff between hassle and accuracy:-

1. Fix it at 0. This is the mirror of the PRAM model which sets the computation cost at 1 and makes communication cost 0!
2. Let the user estimate it (in units of seconds, or number of memory accesses, arithmetic operations, etc).
3. Cycle count the assembly code.
4. Measure the times on the fly. This is only appropriate when developing on the target platform and not multitasking or multithreading on a single processor.

5. Measure the important times with a profiler off line.

Option 1, ignoring computation altogether, yields graphs showing the total communication time for an algorithm on a machine, which may be useful in itself as it shows how computation time must fall in order to make use of the machine. Option 2 is surprisingly useful. The programmer adds calls to a "`compute(N)`" macro which adds N "time steps" to the local simulated time, where a "time step" is the time taken to perform an arithmetic operation. This time is highly variable because of the influence of the memory hierarchy, but may be bracketed between likely limits (e.g. between 1 and 10 microseconds). This time step can be given as a parameter to the reverse profiler, so one may check how a design fares when given minimum expected compute step time and maximum expected communications time (the worst case for parallel algorithm scalability). Saavedra-Barrera (1989) describes characterisation routines for measuring the performance of different classes of operations in Fortran and if such figures were generally available for sequential code it would make parallel design easier.

Cycle counting of assembler code (option 3) is the preferred choice of the simulators. This technique has been shown to yield very accurate time estimates (Brewer, 1991). It involves an extra compilation stage, with the assembly code for the application being interpreted and augmented by a routine which inserts instructions to update a global cycle count after each basic block. Since the number of cache misses may lead to an order of magnitude variation in the execution time, a cache model is required for such simulators. This technique also requires augmented versions of all libraries used.

Experience using the Proteus `augment` tool indicated that though the technique works, it is too time consuming and awkward for quick estimates of compute time. It is also a "black box" approach and it it hard to know how reliable the estimates will be.

Option 4, measuring the compute times on the fly, is tricky on a multi-tasking system. Some multi-threading libraries provide "virtual timers" which only measure compute time consumed by the current thread, but these are not generally available. In any case, the compute times would have to be scaled for the target architecture.

The final option, profiling important subroutines on the target system and feeding the numbers back into the reverse profiler yields the most believable numbers.

5 EXAMPLE

This section illustrates results obtained by using reverse profiling with the `outer` routine from the Cowichan suite of problems (Wilson, 1994).

`outer` is given a set of N (x, y) coordinates and computes the distance of each point from every other point. These distances are stored in a $N \times N$ matrix. Since the distance from point A to point B is the same as from B to A, the matrix is symmetric about its diagonal. For N points, $N^2/2 - N$ distance computations are needed. The diagonal values of the matrix are all set to N times the maximum off-diagonal value. The routine also generates a real vector of distances of each point from the origin.

The MPI implementation of the routine generates the matrix and vector as distributed data structures, with an equal number of rows on each processor. Each process calls `MPI_Allgather` to take a local copy of the input points. It then computes the local section of the vector and the matrix, performs an `MPI_Allreduce` to determine the maximum distance across the matrix and fills the local section of the diagonal.

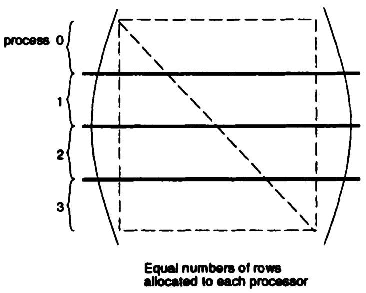


Figure 2 `outer` : matrix distribution across 4 processors

Each process computes the distances for all the matrix positions below the diagonal as well as those above it, thus doing twice the amount of work necessary, but not requiring any extra communication.

The routine is thus very simple, yet it is not trivial to work out how fast it will run on a range of problem and machine sizes.

A characterisation of the EPCC's implementation of MPI on the Cray T3D was generated using the routines described above. The `outer` routine was linked with the reverse profiling library on a workstation running the LAM implementation of MPI. The routine was then run on the workstation varying the number of processes and data sizes to obtain predictions of how it would perform on the T3D.

In the code, an example of one "compute step" is:

```
matrix[r - matrix.local_displ()][c] = d;
```

i.e. it is an extremely crude estimate of the time. A reasonable estimate of the time that this would take on the 150MHz DEC Alpha processors used in the Cray is hard to make without a detailed knowledge of the cache, compiler optimisations, pipelines and main memory latency. A direct execution simulator would work with the assembly code which enables the effect of compiler optimisations to be measured, but still leaves the pipelines and memory hierarchy to be modelled (which is possible, but not convenient).

The time for a basic compute step was left as a parameter and varied from $100ns$ up to $1us$ to see the effects on speedup, estimating that the line of code above (which includes a function call, a subtraction, two array indexing operations and a store to memory) would take between 15 and 150 cycles on a processor with a $6.6ns$ cycle time.

Figure 3 shows the measured and predicted speedups, which correspond reasonably with a compute step set between $0.1us$ and $1us$.

For this example reverse profiling gives a reasonable prediction of the speedup as long as the compute time can be estimated. It also allows "what if" experiments on a design to see how it can be expected to behave.

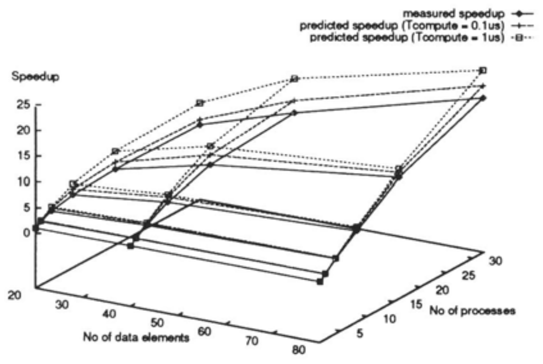


Figure 3 outer : predicted and measured speedups on the Cray T3D

6 CONCLUSIONS

Reverse profiling offers a very quick and easy method of performance prediction for MPI programs. Unlike simulation techniques it builds directly upon the full and complete MPI libraries available now. It doesn't attempt to handle non-determinism but this is the area in which existing profilers and simulators produce the least believable results. It works with any MPI implementation which provides the standard profiling interface, so predictions may be performed in parallel.

It is intended to complement rather than replace analytical approaches; making the model available to programmers allows pencil and paper analysis where appropriate.

The most important next stage is to obtain feedback from users to judge whether the current balance between simplicity and accuracy is appropriate. Work is also currently in progress investigating whether a similar technique could be applied to a shared memory programming model.

7 ACKNOWLEDGEMENTS

Thanks to Marcus Marr for suggestions on the MPI characterisation routines and also to the anonymous reviewers for their detailed and constructive comments.

REFERENCES

Brewer, E.A., Dellarocas, C.N., Colbrook, A. and Weihl, W.E. (1991) PROTEUS: A high performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science.

Brewer, E.A. and Weihl, W.E. (1993) Developing parallel applications using high-performance simulation. In *Proceedings of 1993 Workshop on Parallel and Distributed Debugging*. San Diego, CA.

Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R. and von Eicken, T. (1993) LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. San Diego, CA, May 1993.

Dickens, P.M., Heidelberger, P. and Nicol, D.M. (1993) A distributed memory LAPSE: Parallel simulation of message-passing programs. Technical Report NAS1-19480, NASA Langley Research Center, Hampton, VA 23681.

Driscoll, M.A. and Daasch, W.R. (1995) Accurate predictions of parallel program execution time. *Journal of Parallel and Distributed Computing*, 25(1).

Message Passing Interface Forum (1995) MPI: A Message Passing Interface. Technical report, University of Tennessee.

Foster, I. (1994) *Designing and Building Parallel Programs*, chapter 3. Addison-Wesley. Available online at http://www.mcs.anl.gov/dbpp/.

Howell, F.W., Williams, R. and Ibbett, R.N. (1994) Hierarchical Architecture Design and Simulation Environment. In *MASCOTS '94: Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*.

Nupairoj, N. and Ni, L.M. (1995) Benchmarking of multicast communication services. Technical Report MSU-CPS-ACS-103, Michigan State University.

Saavedra-Barrera, R.H., Smith, A.J. and Miya, E. (1989) Machine characterisation based on an abstract high-level language machine. *IEEE Trans. on Comp.*, 38(12), 1659–1679.

Sarukkai, S.R. (1994) Scalability analysis tools for SPMD message-passing parallel programs. In *MASCOTS '94: Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*.

Wabnig, H. and Haring, G. (1995) Performance prediction of parallel systems with scalable specifications - methodology and case study. *Performance Evaluation Review*, 22(2).

Wilson, G.V. (1994) Assessing the Usability of Parallel Programming Systems: The Cowichan Problems. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*. Birkhäuser Verlag AG, April 1994.

8 BIOGRAPHY

Fred Howell received his BSc and MEng degrees in Microelectronic Systems Engineering from the University of Manchester Institute of Science and Technology in 1992. He is currently a PhD student at the University of Edinburgh Department of Computer Science where his research interests include the design of parallel hardware and software. He has been funded by EPSRC and by Digital (Scotland) Ltd.

PART TWO

Project Reviews

22

SEMPA: Software Engineering Methods for Parallel Scientific Applications

P. Luksch, U. Maier, S. Rathmayer, M. Weidmann
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Institut für Informatik, Technische Universität München
D-80290 München
e-mail: {luksch, maier, maiers, weidmann}@informatik-tu.muenchen.de
WWW: http://wwwbode.informatik.tu-muenchen.de/
Tel.: (089)2105-8164; Fax: (089)2105-8232

Abstract
SEMPA is an interdisciplinary project that brings together researchers from computer science, mechanical engineering and numerical mathematics. Its central objective is to develop new software engineering (SWE) methods for (distributed memory) parallel scientific computing. SEMPA is being funded by the BMBF*.

1 MOTIVATION

In many applications, fluid simulations are required because experiments are either impossible (such as in the case of climate modeling) or are too expensive. Today, the main factor that limits the use of simulation is run-time. Only parallel processing together with efficient numerical algorithms can achieve the performance that is necessary to enable more wide-spread use of simulation. Providing the necessary computational power will make simulations feasible in many areas where they would require unrealistic run-times today. In mechanical engineering, productivity can be considerably increased if flow simulations, which today have to be run as batch jobs overnight, could be run interactively from a CAD program.

Parallel processing has developed successfully in the research area over the last years. Now, as there are standardized message passing interfaces, such as PVM [GBD+94], MPI [MPI94] etc., portable software can be developed for a wide range of hardware platforms – from (heterogeneous) networks of workstations (NOWs) to high-end massively parallel systems (MPPs). Since even small companies usually have a number of workstations connected by a local area network (LAN), developing parallel software on a commercial basis is becoming an attractive option.

However, experience has shown that software development for parallel systems still is much less productive than writing sequential programs. One reason for this is that there are no adequate tools for designing and analyzing

*Federal Department of Education, Research and Technology parallel software. In addition, there are no software engineering (SWE) methods that address the problems related to parallelism such as synchronization issues, deadlocks and non-determinism. Finally, there is only very little support for the software engineer who is faced with the problem of understanding and existing program in order to parallelize it for execution on a distributed memory multiprocessor.

2 PARTNERS

LRR-TUM (Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München). LRR-TUM is in charge of project management. Our research focuses on
- multiprocessor architectures
- tools for designing and analyzing parallel programs
- parallel and distributed applications
- distributed shared memory systems.

Advanced Scientific Computing GmbH (ASC) , Holzkirchen. ASC is developing and marketing the CFD simulation package **TASCflow** which solves the Navier-Stokes equations in 3d space. **TASCflow** is used in many companies and universities for simulating flows in a wide range of applications [TUG95].

GENIAS Software GmbH, Neutraubling near Regensburg. The company markets a number of software packages for NOWs and MPPs. They have developed the batch queuing system **CODINE** on NOWs, which will be the basis for the resource to be developed in SEMPA.

Institut für Computeranwendungen (ICA III), Universität Stuttgart. ICA's research is focused on robust multigrid methods for a wide range of problems including computational fluid dynamics, flow in porous media and computational mechanics. They have developed the software tool-box UG, which simplifies the adaptive solution of partial differential equations on unstructured meshes in two and three dimensions.

3 OBJECTIVES

Software Engineering Methods. In parallel scientific computing, software engineers usually are faced with an existing program or with existing modules, typically written in FORTRAN 77, which they are expected to parallelize for execution on a distributed memory multiprocessors (NOW or MPP). Therefore the focus of SWE is on the following topics:
- Analysis of complex software systems.
- Approaches to Parallelization that are specific to certain classes of scientific applications
- Standards for documentation and program development
- Portability: cover a wide range of hardware platforms ranging from (low-end) NOWs to high performance MPPs.
- Modularity and Re-usability.
- Concurrent Software Engineering: coordinate the work of programmers from different disciplines and institutions

Parallelization of TASCflow. The software package solves the Navier-Stokes equation in 3d space on unstructured grids using a finite volume discretization and an algebraic multi-grid solver. The program is written in FORTRAN 77 and has about 113,000 lines of code.

Load Balancing and Resource Management. A resource manager is being developed, which basically is a batch

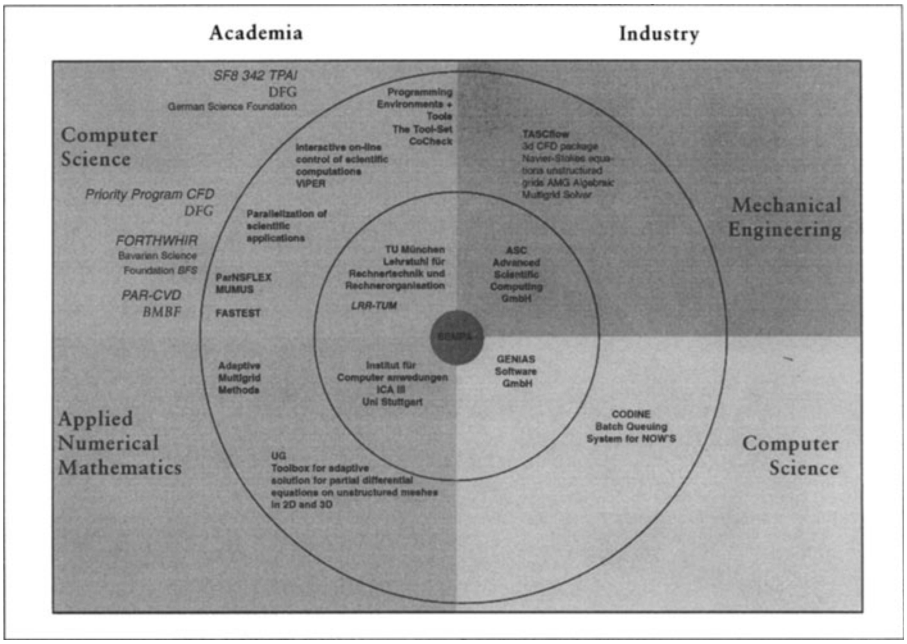

Academia
Industry
SF8 342 TPAI
DFG
German Science Foundation
Computer Science
Priority Program CFD
DFG
FORTHWHIR
Bavarian Science Foundation BFS
PAR-CVD
BMBF
Applied Numerical Mathematics
Programming Environments + Tools
The Tool-Set CoCheck
Interactive on-line control of scientific computations VIPER
Parallelization of scientific applications
ParNSFLEX
MUMUS
FASTEST
Adaptive Multigrid Methods
TU München Lehrstuhl für Rechnertechnik und Rechnerorganisation
LRR-TUM
SEMPA
Institut für Computer anwedungen ICA III Uni Stuttgart
UG
Toolbox for adaptive solution for partial differential equations on unstructured meshes in 2D and 3D
TASCflow
3d CFD package Navier-Stokes equations unstructured grids AMG Algebraic Multigrid Solver
Mechanical Engineering
ASC Advanced Scientific Computing GmbH
GENIAS Software GmbH
CODINE Batch Queuing System for NOW'S
Computer Science

Figure 1 partners involved in the project

queuing system for parallel applications running on NOWs. The individual processes of the application are dynamically assigned to available processors (i.e. workstations). The resource manager will support load balancing by providing appropriate resource usage information and a mechanism to migrate processes from one workstation to another.

The result of SEMPA will be

- a collection of SWE methods that have been approved in practice,
- a prototypical implementation of the parallel version of **TASCflow**,
- a prototype of a resource and load manager for batch execution of parallel applications.

Upon completion of the research project, our industrial partners intend to develop further the prototypes of the parallel CFD package and the resource manager towards products that can be marketed commercially.

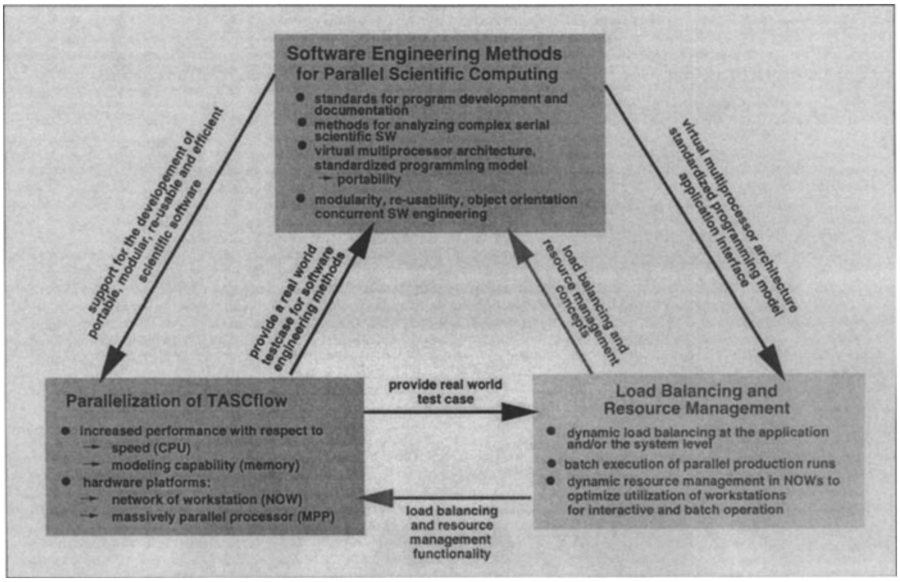

Software Engineering Methods for Parallel Scientific Computing
• standards for program development and documentation
• methods for analyzing complex serial scientific SW
• virtual multiprocessor architecture, standardized programming model → portability
• modularity, re-usability, object orientation concurrent SW engineering
support for the developement of portable, modular, re-usable and efficient scientific software
provide a real world testcase for software engineering methods
load balancing and resource management concepts
virtual multiprocessor architecture standardized programming model application interface
Parallelization of TASCflow
• increased performance with respect to
→ speed (CPU)
→ modeling capability (memory)
• hardware platforms:
→ network of workstation (NOW)
→ massively parallel processor (MPP)
provide real world test case
load balancing and resource management functionality
Load Balancing and Resource Management
• dynamic load balancing at the application and/or the system level
• batch execution of parallel production runs
• dynamic resource management in NOWs to optimize utilization of workstations for interactive and batch operation

Figure 2 project objectives and their interactions

4 PROGRESS REPORT

The project has started in April, 1995. Up-to-date information about progress as well as project reports and publications related to SEMPA are available via WWW (URL http://wwwbode.informatik.tu-muenchen.de/parallelrechner/applications/sempa/). In the subsequent section, we summarize the results achieved so far.

4.1 Analyzing the sequential program

The first step in parallelizing **TASCflow** has been to acquire the necessary understanding of the algorithms it uses and their implementation. ASC and LRR-TUM have been organizing a series of meetings, covering the following topics (in that sequence):

1. basics of CFD, i.e. the governing equations and their physical interpretation, discretization methods, and numerical methods for solving the system of linear equations that results from discretization.
2. a global overview of the code structure and the main data structures.
3. a more detailed review of **TASCflow**'s main modules, stepping through each module subroutine by subroutine.

Each meeting started with a presentation by ASC, which was followed by a discussion. At LRR-TUM, we documented our view of what we had learned in a meeting in form of an internal report, which then was reviewed by ASC. This procedure has proved to be an efficient way for know-how transfer between groups from different disciplines, since it has helped to identify and fix sources of misconception very early and quickly improved our understanding of each other's terminology and point of view.

As a final step, a framework has to be set up that defines a standard for documenting the design of the sequential program from the computer science point of view.

4.2 A Concept for Parallelizing TASCflow

Based on the insight gained from analyzing the program structure, a parallelization concept has been defined and documented [Luk95]. SEMPA follows a two-level concept of parallelism.

On the top level the SPMD model is used. The sequential algorithm† is replicated in multiple processes each of which operates on a partition of the problem description. An additional master process is used for program set up and for doing I/O. Using parallel I/O systems, which are available for a number of platforms, is being considered, too.

Partitioning is done node-based, i.e. the nodes of the (unstructured) grid are divided into disjoint sets. We use a public domain graph partitioning package (MeTiS [MET95]) for assigning nodes to partitions.

Below the SPMD level of parallelism, parallelization is considered at the level of processing nodes. Each replicated worker of the SPMD model can be further parallelized into a number of concurrent threads (light-weight processes having access to shared memory). This second level of parallelism can make use of multiple CPUs per processing node as they are available in new MPPs or workstations.

4.3 Interactive and automatic Parallelization Tools

The parallelization of an existing program, especially if it is complex and has been developed by many engineers over a long time, is a quite difficult and error-prone task.

Research projects as well as commercial efforts during the last years have been dealing with this problem. Most available tools are source-code analyzers for FORTRAN 77 programs which parallelize according to the SPMD model. One of those tools has already been subject of investigation within SEMPA. It is the quite sophisticated interactive an automatic parallelization tool FORGE [Res95]. There the most significant loops are identified by either using profiling information, or checking the code for the deepest loop nestings. Once the loops have been chosen, the arrays referenced inside of them are partitioned and distributed according to the partitions. The parallel processes then run the same program but only on a subset of the partitioned data structures following the so-called *owner computes* rule.

The advantage of these tools is that the user can get a better understanding of the program that he is about to parallelize. He also is taken off the burden to explicitly program message passing code. On the other hand he anyhow has to have a good understanding of how message passing really works because the tools are not yet at a point where they can produce efficient code. Neither can they really work on very complex packages as for example TASCflow.

4.4 New Languages

Moving from FORTRAN 77 to newer programming languages meets the requirements of modern computer architectures, programming paradigms, and software engineering aspects. Fortran 90 for example offers not only more

† augmented by additional code for communication and synchronization

complex data structures and data encapsulation but also provides language constructs (array operations) for concurrent execution. The latest of all FORTRAN evolutions, High Performance Fortran, additionally has constructs for explicit data distribution as well as constructs for expressing concurrency.

FORTRAN 77 compilers produce fast object code and numerous numerical programming libraries are available due to its long time of existence. Object oriented design is still uncommon in scientific computing because the compilers (e.g. for C++) do not yet generate optimized code that is comparable to the one generated by FORTRAN 77 compilers. However, the fundamental ideas of object oriented programming – objects, class-hierarchies and polymorphism – are of great advantage to modern software engineering and can help to overcome the gap between the code development and its concept.

In SEMPA, we have decided that rewriting TASCflow as a whole in an object-oriented language is infeasible due to manpower restrictions. Instead, we have selected a module of reasonable size for implementation in Fortran 90, C++ (and possibly other languages) to demonstrate the integratability of object-oriented techniques to a scientific application, and to evaluate the appropriateness of these languages for our purposes.

REFERENCES

[GBD$^+$94] *Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam.* "PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing". MIT Press (1994). www: http://www.netlib.org/pvm3/book/pvm-book.html.

[Luk95] *Peter Luksch.* A Concept for Parallelizing TASCflow. SEMPA-Report SEMPA-TUM-95-05, Technische Universität München, Institut für Informatik (September 1995). www: http://wwwbode.informatik.tu-muenchen.de/archiv/Projektberichte/SEMPA/ws-aug-95.ps.gz. draft version.

[MET95] "METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System". George Karypis and Vipin Kumar, University of Minnesota (1995).

[MPI94] MPI: A Message Passing Interface Standard. Technical report University of Tennessee, Knoxville, Message Passing Interface Forum (May 1994).

[Res95] *Applied Parallel Research.* "The FORGE Product Set". Applied Parallel Research Inc., 550 Main Street, Placerville, CA 95667 (February 1995).

[TUG95] 3rd TASCflow User Conference – Presentations. Tech. Report ASCG/TR-95-04, Advanced Scientific Computing GmbH, Aying (May 1995).

23
EPOCA: status and prospects

S. Donatelli[a], *N. Mazzocca*[b] *and S. Russo*[b]

[a] Dipartimento di Informatica, Università di Torino

Corso Svizzera 185, 10149 Torino - Italy

Tel.: +39 (0)81 7429246 Fax: +39 (0)81 7429 E-mail: susi@di.unito.it

[b] Dipartimento di Informatica e Sistemistica, Università di Napoli "Federico II"

Via Claudio 21, 80125 Napoli - Italy

Tel.: +39 (0)81 7682893/5 Fax: +39 (0)81 7683186 E-mail: mazzocca,russo@nadis.dis.unina.it

Abstract

In this paper we present a review of the aims, achievements and prospects of the EPOCA project. EPOCA is a Petri net based system for performance evaluation and analysis of concurrent applications. Research issues, current outcomes and future directions of the project are described.

Keywords

Parallel software engineering, Petri nets, CSP, performance evaluation

1 PROJECT AIMS AND MOTIVATIONS

EPOCA (Environment for Performance evaluation and analysis Of Concurrent Applications) is a CASE system that supports the development, the qualitative analysis and the predictive performance analysis of parallel and distributed programs. EPOCA is a joint project of the Performance Evaluation group at the Department of Computer Science of the University of Torino, and of the Parallel Architectures group at the Department of Computer Science and Systems of the University of Napoli. The project had two main objectives:

- the investigation of issues in the inclusion of Petri net based formal techniques for both program validation and performance evaluation into parallel software development;
- the definition and the construction of related appropriate CASE support tools.

To meet these goals, a methodology has been defined, centred on the use of a class of timed Petri nets, namely Generalized Stochastic Petri Nets (GSPNs). GSPNs (Ajmone-Marsan, 1984) have been chosen because they are a formalism suited to study time independent (correctness), as well as time dependent (performance) properties of concurrent programs, and because they allow performance figures to be computed either by solving the Markov chain isomorphic to a GSPN, or, for large models, via simulation (Donatelli, 1994-a),

The methodology iterates through the following steps, until an implementation is obtained, which is correct and meets desired performance requirements:

- application implementation in a C-based CSP-like language;
- construction of a qualitative GSPN representation of the program;
- program behaviour analysis and validation via net analysis;
- introduction of program quantitative parameters in the untimed model;
- definition and computation of performance indices.

2 PROJECT OUTCOMES

The project has resulted at this stage in the following outcomes:

- a methodology for modelling and analysing concurrent programs;
- an integrated prototype CASE support system for the development, modelling and analysis phases;
- scientific publications: the EPOCA bibliography below lists the major journal and international conference papers and technical reports;
- scientific collaborations: they have been established with other groups active in the area of parallel software engineering, to cover research issues at the intersections with other projects.

In the following, the features of the EPOCA prototype system and the experience with it are summarized.

3 THE CASE SYSTEM EPOCA

In EPOCA results and proofs about a concurrent program are derived from results and proofs on a GSPN model of it. This approach is similar to the one of Shatz and Cheng for Ada (Shatz, 1987), which is limited to qualitative analysis. The EPOCA system results from the integration of different, and in same way complementary, experiences of two research groups, which resulted in the development of two tools: DISC (Iannello, 1990), a message-passing language developed at the University of Napoli, and GreatSPN (Chiola, 1991), a GSPN editing and analysis tool, developed at the University of Torino. The main functionalities of EPOCA are: translation of DISC programs into GSPN models, analysis of time-independent properties (based on structural or state space analysis), and program quantitative characterization through automatic definition and computation of net performance indices. Of course the overall EPOCA environment includes also all the functionalities of DISC (makefile generator, compiler, linker tools, run time support, monitor, post-mortem analyzer and window-based interface), and those of GreatSPN (graphical GSPN editor, net analysis tools).

EPOCA allows to build a model of the process interactions. Program flow control statements are modelled only if they contain inter-process communications. Variables are

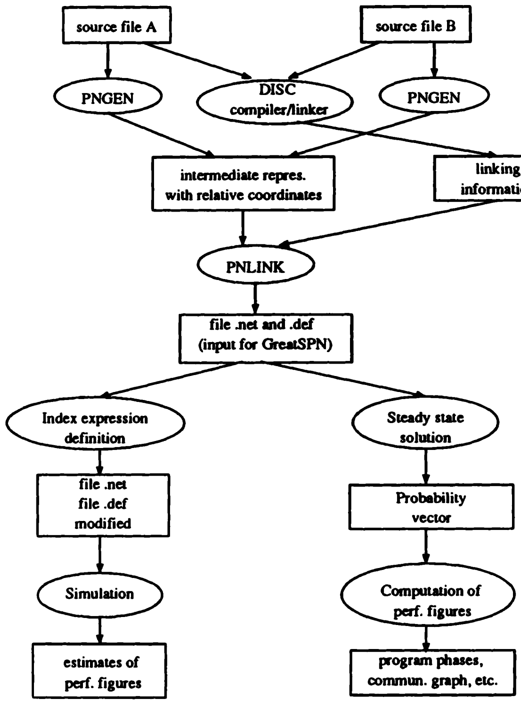

source file A
source file B
PNGEN
DISC compiler/linker
PNGEN
intermediate repres. with relative coordinates
linking information
PNLINK
file .net and .def (input for GreatSPN)
Index expression definition
Steady state solution
file .net file .def modified
Probability vector
Simulation
Computation of perf. figures
estimates of perf. figures
program phases, commun. graph, etc.

Figure 1 The program translation and analysis process in EPOCA.

not modelled automatically. Purely sequential parts of the code are collapsed into timed transitions, whose timing parameter represents the code execution time. Using this level of abstraction, the GSPN model is appropriate to analyze qualitative and quantitative properties, avoiding to deal with the detailed modeling of the sequential part of the code. Since the application does not need to be complete, for instance only the basic structure of process communication and activation may have been already coded, EPOCA allows the evaluation of certain design decisions without requiring the full implementation to be available.

Figure 1 summarizes the main features of the translation and solution process applied to a DISC program (assumed to be composed of 2 files), showing the role of the main EPOCA tools. The PNGEN module compiles each source file into an intermediate representation, that contains the translation of each process: all cross-references still have to be solved. The PNLINK module combines the intermediate files to produce the global GSPN model (defined by a ".net" and a ".def" file). The final format suitable for GreatSPN, generated by PNLINK, consists of the topological description and of the graphical layout of the net. This is useful for net visualization, animation and further editing. Proper tools provide an automatic definition of the GreatSPN expression of the desired performance indices. The model is then solved either exactly or approximately through a discrete event simulator.

4 USING EPOCA

The analysis of an application in EPOCA can start from a program skeleton (prototype), that defines the synchronization and the interactions between processes, abstracting from the algorithmic details. This gives the possibility to investigate the properties of the application using an incremental validation approach; for instance, the presence of a deadlock should be detected independently and before the evaluation of performance indices, that requires the complete estimate of the quantitative parameters.

After the automatic derivation of the GSPN model, the analyst can start performing the qualitative program analysis. This can be done using algebraic techniques, or executing the net model, or by the inspections of the reachability graph. The *algebraic techniques* provide information on the correctness and the behavior of the system. The *reachability graph* allows to investigate the dynamic evolution of the concurrent program through its state space, and it is the basis for constructing a performance model.

The performance figures computed by EPOCA can be grouped in three classes: phases, communication costs and interference costs. A *program phase* is a period of the execution in which a given set of processes is active. EPOCA computes the set of phases, the duration of each phase, the maximum/average number of processes concurrently active (degree of parallelism). *Communication costs* are used to study quantitatively the communication profile of the application: for instance, they allow to build the program communication graph. *Interference costs* express the probability of two processes requiring the CPU at the same time. All these metrics provide a quantitative characterization of the program running on the best possible platform.

The EPOCA methodology and tools have been applied to several case studies, representative of different common application classes. Table 1 reports a survey of the experiences with EPOCA, summarizing for each class the main goals of the program analysis, and the metrics computed by the EPOCA tools. These experiences show that EPOCA provide an invaluable aid for applications where formal analysis is of fundamental importance, such as safety critical ones, and that it is extremely useful for process control and distributed client-server applications, for which the dependency from input data is very well described by a stochastic model. The tools are also useful to reason on distributed algorithms, while for parallel numerical applications the Petri net-based approach to performance prediction gives results comparable to those of other methods (analytical models and monitoring techniques), but in many cases it is less effective.

Readers interested in getting a deeper insight in the different issues addressed in the project can refer to the EPOCA bibliography below. Balbo (1992) presents the net translation rules for the CSP constructs, while Donatelli (1994-b) describes the translation from DISC to GSPN. Balbo (1994) defines the program performance indices, and shows how they can be computed on the GSPN model. Donatelli (1994-a, 1994-b, 1996), Mazzeo (1996) and Jelly (1995) contain the description of some case studies (see Table 1). Donatelli (1994-a) discusses also the advantages and disadvantages of having chosen the GSPN formalism as the model formalism of EPOCA. Donatelli (1994-b) gives details on the architecture of the tool, while the role of EPOCA as a tool for computer aided distributed software engineering is discussed by Donatelli (1996).

Table 1 A survey of the experiences with EPOCA.

Application class	Description	Analysis' goals	Computed parameters
Client-server	Inter-departmental administrative computing center (Donatelli, 1996)	Deadlock freedom, program comprehension, bottleneck detection, performance tuning (with varying data transfer rates and remote data access probabilities)	*Qualitative*: deadlock states, reachable states, net animation. *Quantitative*: service time, communication graph, efficiency, degree of parallelism
Process control	Monitoring system (stochastic dependency from input data) (Balbo, 1994)	Program correctness and performance	*Qualitative*: deadlock and reachable states. *Quantitative*: mean service time, comm. graph, efficiency, degree of parallelism
Safety-critical systems	Railway transport system	Deadlock freedom, program comprehension, safety analysis, performance analysis	*Qualitative*: deadlock and reachable states, net animation. *Quantitative*: probab. of hazard states
Distributed algorithms	Circuit simulator (Balbo, 1992)	Program correctness	*Qualitative*: deadlock detection
Parallel numerical algorithms	FFT (Donatelli, 1994-a)	Comparison of data partitioning and communication strategies; program performance as a function of the data transfer rate and of the node computing power	*Qualitative*: deadlock detection. *Quantitative*: degree of parallelism, program phases, communication graph, completion time, CPU utilization

5 CONCLUSIONS AND FUTURE DIRECTIONS

The EPOCA project has addressed a problem which is perceived as increasingly important in the concurrent software engineering community, namely the definition of formal techniques and tools to support the analysis and the validation of the system implementation (i.e., the program). In EPOCA a methodology has been proposed, based on the use of the GSPN formalism. The project has shown that this methodology can be fully supported by automatic tools, and that an effective approach to do that is to integrate existing CASE tools for program development and net analysis. Although EPOCA has considered a specific CSP-like language, most of its features are independent of the details of the programming language used for system implementation. The project has pointed out that programs should have specific characteristics - such as a clear separation of structural and behavioural aspects - for static analysis tools to be capable of modelling them automatically.

The experience in EPOCA proved attractive and promising. This has suggested to extend the approach to the introduction of verification and performance evaluation techniques into the *design* refinement phase of parallel software development. Work has begun to investigate the benefits of introducing the EPOCA approach into the PARSE methodology (Russo, 1995). PARSE (Gorton, 1995) is a design methodology is based on a graphical notation, that enables the developer to describe a parallel software system in terms of a collection of concurrent components interacting via message passing. The transformation from the PARSE design domain to the Petri net domain provides two fundamental advantages: (a) it provides formal support for design verification, (b) it allows to animate the system design specification, by executing the net. The refinement steps, until the analysis reveals that the design is error-free and that it can meet the performance requirements, will provide the software engineer with increased confidence in the final quality of the design, before proceeding to the implementation.

REFERENCES

Ajmone Marsan, M., Balbo, G. and Conte, G. (1984) A class of generalized stochastic Petri nets for the performance analysis of multiprocessor systems. *ACM Trans. Comp. Systems*, 2(1).

G. Chiola. (1991) GreatSPN 1.5 software architecture. *Proc. 5th Int. Conf. Modelling Techniques and Tools for Computer Performance Evaluation*, Torino, Italy.

Gorton, I., Gray, J. and Jelly, I.E. (1995) Object-Based Modelling of Parallel Programs. *IEEE Parallel and Distributed Technology*, **3(2)**.

Iannello, G., Mazzeo, A., Savy, C. and Ventre, G. (1990) Parallel software development in the DISC programming environment. *Future Generation Computer Systems*, **5(4)**, 365–372.

Shatz, S.M. and Cheng, W.K. (1987) A Petri net framework for automated static analysis of Ada tasking behaviour. *The Journal of Systems and Software*, **8**, 343–359.

EPOCA BIBLIOGRAPHY

Balbo, G., Donatelli, S. and Franceschinis, G. (1992) Understanding parallel program behavior through Petri net models. *Journal of Parallel and Distributed Computing*, **15(3)**, 171–187.

Balbo, G., Donatelli, S., Franceschinis, G., Mazzeo, A., Mazzocca, N. and Ribaudo, M. (1994) On the computation of performance characteristics of concurrent programs using GSPNs. *Performance Evaluation*, **19**, 195–222.

Donatelli, S., Franceschinis, G., Ribaudo, M. and Russo, S. (1994) Use of GSPNs for concurrent software validation in EPOCA. *Information and Software Technology*, **36(7)**, 443–448.

Donatelli, S., Franceschinis, G., Mazzocca, N. and Russo, S. (1994) Software architecture of the EPOCA integrated environment. *Proc. 7th Int. Conf. Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS No.794, Springer-Verlag.

Mazzeo, A., Mazzocca, N., Russo, S. and Vittorini, V. (1996) A method for predictive performance evaluation of distributed programs. To appear in *Simulation: Practice and Theory*.

Russo, S., Savy, C., Jelly, I.E. and Collingwood, P. (1995) Petri net modelling of PARSE designs. Joint Tec. Rep. 7/95, Computing Research Centre, Sheffield Hallam Univ. and Univ. of Naples.

24

The PARSE Project

I. E. Jelly[1] and I. Gorton[2]
[1]Computing Research Centre, Sheffield Hallam University,
Sheffield S11 8HD, UK. Email: i.jelly@shu.ac.uk
Tel : +44 (0)114 253 3763, Fax: +44 (0)114 253 3161

[2]CSIRO Division of Information Technology, Locked Bag 17, North Ryde,
NSW 2113, Australia. Email: iango@syd.dit.csiro.au
Tel: +61 (0)2 325 3160, Fax: +61 (0)2 325 3101

Abstract
Within the PARSE project, issues relating to the development of parallel and distributed software are being researched. These include analysis and design techniques, verification of system behaviour, performance evaluation and tool support. This paper reviews the work undertaken in the project, indicates future directions for research and provides a bibliography of key publications.

Keywords
parallel processing, distributed systems, software engineering, EPOCA, PARSE

1 PROJECT AIMS

The PARSE (PARallel Software Engineering) project began in 1991 as a collaborative project between Sheffield Hallam University and two Australian Universities. Its aim has been the development of techniques and tools to support the production of high quality software for a wide range of parallel and distributed systems. The collaboration has been extended to the Universities of Sheffield and Naples, and now includes an industrial partner.

Within the project, research has been focused on the issues involved in the development of parallel and distributed software systems. It brings together a number of projects which are addressing a range of technical issues within the field. These involve the development of appropriate techniques and tools to support good software engineering practice for parallel and distributed systems. The collaboration has revealed a unifying approach to the consideration of different aspects of the development of systems, based round the *integration of pragmatic design engineering principles and formally based techniques.*

Currently parallelism in software systems can be employed either to provide better performance in large scale scientific computations, or to support the development of smaller scale systems where issues of real time constraints, physically distributed control and fault tolerance are involved. In addition, system programmers constructing operating systems and associated low-level system utilities exploit concurrent tasks to share and manage resources amongst users and user programs. However the importance of parallel software is likely to grow in the next decade. Major technological advances in processing power, desktop operating systems, networking and user-interface systems will converge, creating new, high-volume application areas in distributed systems, consumer electronics and multimedia systems. All of these naturally lend themselves to parallel software solutions.

Within the PARSE project the emphasis had been on the development of design principles to promote the construction of efficient, robust and reusable parallel and distributed systems. A general framework is proposed to cover a range of application types and implementation environments. The designer may need to employ different views and techniques within the development process; the importance of providing clear mechanisms for interfacing with these has been recognised from the outset of the PARSE project, and the concept of methods integration is central to our approach. Although primarily aimed at software development, we are currently applying the techniques in relation to hardware/software codesign.

2 PROJECT OUTCOMES

The major outcomes from the project at this stage are:

1. Over twenty journal and international conference papers, and a number of technical reports.
2. Prototype CASE tools for the key phases of PARSE have been developed and demonstrated.
3. Techniques developed as part of the project are being employed for the production of industrial applications, thus providing feedback to the academic partners.
4. The collaborators have been active in the establishment of a research community in the area of parallel and distributed software engineering, and have organised workshops on this topic.

3 KEY RESEARCH ISSUES

3.1 PARSE Methodology

The PARSE methodology supports a systematic design refinement process and provides a graphical approach (PARSE process graphs) to express high level architecture and language independent design abstractions. A staged design methodology has been defined within which the developer moves from a high level abstract view of the system to a more concrete representation of the software. Different application types may require variations on the process: for example, real time systems design requires the introduction of temporal constraints at an early stage in the design process; for safety critical systems, formal verification may be required throughout the development phases.

3.2 Design Notation

The PARSE process graph notation allows developers to describe the system in terms of a hierarchy of interacting components. It is object-based and supports a staged design refinement approach. Process objects interact by message passing on designated communication paths. Both process objects and communication paths are classified according to their role in the system. In addition, information about the dynamic behaviour of the system is captured with the graphical notation by the introduction of path constructors which describe the order of handling of incoming messages by process objects. Process graphs can be systematically supplemented with additional behavioural details by using formalisms such as Petri nets or CSP, or a dedicated Behaviour Specification Language (BSL). Translation from BSL descriptions into Petri net specifications can be fully automated and forms the basis for the work within the EPOCA project.

3.3 Tool Support

Prototype design editors have been constructed to support developers in the construction of process graph/BSL designs, and work on more sophisticated versions of these is proceeding. The emphasis within this work is to ensure that the tools developed will support flexible integration of different techniques: appropriate interfaces must be defined to allow easy access to a range of verification and simulation and programming systems. Hence we are exploring the use of meta-CASE systems to implement the PARSE support tools. In order to bridge between the design and implementation stages of the software development cycle, research into code generation for designs specified in BSL format has lead to the development of prototype code generators. Currently PVM and Occam tools have been implemented and code generators for DISC (Distributed C) under the EPOCA project are under consideration.

3.4 EPOCA Integration

Design validation plays an important role in the development of robust software systems, and the integration of behavioural analysis techniques involving Petri nets has been explored. Original work within the PARSE project demonstrated the potential for use of Petri net analysis, and recent collaboration with the EPOCA project team has provided a mechanism for this.

Under the EPOCA project, techniques have been developed for the analysis of parallel software systems by means of stochastic Petri nets. This resulted in the integration of the stochastic Petri net toolset GreatSPN with the DISC (DistributedC) programming environment. DISC allows distributed applications to be implemented and executed on a network of workstations. DISC programs can be automatically analysed by the use of GreatSPN.

Within the PARSE project further integration is planned. We have demonstrated how PARSE BSL descriptions can be translated either into DISC code, or directly to GreatSPN format. The design can therefore be subjected to analysis to provide initial feedback to the developer, and further analysed at the code stage. Work is currently underway to automate this translation and provide an interface to GreatSPN, a toolset based on stochastic Petri nets. This supports both qualitative and quantitative analysis, and introduces scope for performance prediction and modelling at the design stage.

3.5 Client-Server Behaviour Modelling

The production of deadlock free software is a primary consideration in the development of many parallel systems. Recent work has investigated how designs can be constrained to facilitate deadlock detection without having to explore the complete state space of the program's execution.. Client-server behaviour modelling involves the development of a design which conforms to certain guidelines with respect to the type of interactions permitted between process objects. The result is a software design which can be directly verified as deadlock free without recourse to analysis using a formalism such as Petri nets. A simple graphical notation has been developed to represent client-server relationships within a software design. This notation can be used in association with PARSE process graphs, by mapping client-server behaviour graphs onto process graph descriptions to provide full design information for the developer. This approach would seem to be especially promising for complex, safety-critical systems.

3.6 Object-Oriented Analysis Techniques

The extent to which traditional analysis techniques offer a foundation for the development of parallel and distributed software systems forms the basis of our work in this area. Original work has suggested that object-oriented analysis techniques such as OMT could be used in association with PARSE design methods. We are now carrying out an evaluation of the role of different analysis techniques within the development framework for parallel and distributed systems.

3.7 Hardware/Software Codesign

Within the PARSE project we are researching techniques to support high level hardware/software codesign for a range of low power, hand held devices with stringent design constraints. Initial work has indicated that the PARSE design approach provides an appropriate framework for implementation independent design descriptions. Process graphs are used to specify the system architecture at a high level of abstraction; and BSL descriptions developed to describe the interactions between components. Translations can then performed from the BSL descriptions into appropriate hardware description languages, eg VHDL, and refined software design specifications. We are researching the introduction of constraint representation and partitioning strategies using process graphs and BSL.

3.8 Design of Dynamic Distributed Systems

The original PARSE process graph notation only supports dynamic process creation in a very limited manner, through the use of process object replication. However, for many applications for distributed computing platforms, this level of design specification is insufficient. We have therefore extended the core notation to cater for dynamic process object and communications path creation, and subsequent destruction. The extensions are currently being utilised and evaluated in the design of distributed interactive multimedia applications, with encouraging results.

3.9 Case Studies and Industrial Trials

In order to validate the PARSE approach, the methodology has been applied to a number of significant software developments, including a parallel logic language run-time support system, a database engine and a transport protocol for high speed networks. More recently the PARSE design method and process graph notation has been used in the development of an industrial real time embedded control system. Combined with the client-server modelling techniques it proved to be an effective and efficient method of construction deadlock free, concurrent software. In addition the notation was found to support team working by providing accessible design documentation.

3 FUTURE DIRECTIONS

The following areas are currently under investigation:
Performance prediction: The development of appropriate simulation tools to support early performance prediction is under investigation. This work will complement the performance evaluation techniques available within the PARSE-EPOCA integration.
Client-server modelling: Future work is planned which will consider the application of these design techniques to a wide range of realistically sized systems. We shall explore the relationship with formal modelling techniques with a view to providing automated support for design verification within which the state explosion problem can be controlled.
High-level design abstractions: We are looking to develop high-level design templates that designers can reuse in different applications.
Designing for distributed object systems: Software development techniques for distributed systems has lead to the definition of different common object co-ordination models to support interaction between objects. We are looking at high level design techniques which allow designers to use these common object models in a secure and effective manner.
Training in parallel and distributed design practice: We are exploring the introduction of the high level design techniques developed during the PARSE project into both academic curricula and industrial training courses.

4 SUMMARY AND COLLABORATION OPPORTUNITIES

The PARSE project currently involves academic staff and students in a number of institutions and industrial partners. It is an open collaboration aimed at providing a forum for exploration of issues relating to parallel and distributed software development. The intention is to support a number of research directions within the project framework, building on existing co-operation to ensure that the high quality of the work is maintained.

We welcome further collaboration - both academic and industrial. We believe that there is a growing range of applications which demand the use of the type of techniques we have been investigating, and we are keen to explore their use in the development of real systems. More information can be found on the PARSE Web Page: http://www.dcs.shef.ac.uk/~prc/parse.html

5 PARSE BIBLIOGRAPHY

5.1 Selected Journal and Conference Papers

Gorton, I., Jelly, I.E. and Gray, J P (1993) PARSE: A Software Engineering Methodology for Parallel Program Design, *in Proc IEEE Int Parallel Processing Symposium*, April 1993, Newport Beach, CA, USA, IEEE Press

Donatelli, S., Franceschinis, G., Mazzocca, N. and Russo, S. (1994) Software Architecture of the EPOCA Integrated Environment *in Proc 7th Int Conf on Modelling Techniques and Tools for Computer Performance Evaluation*, May 1994, Vienna, Austria, Springer Verlag LNCS 794, pp 335-352, Jelly, I.E. and Gorton, I. (1994) Software Engineering for Parallel Systems *in Information and Software Technology Journal,* Vol. 36, No 7, pp 381-396

Knowles, C. and Collingwood, P. (1994) Parallel Software Development using an Object-Oriented Modelling Technique *in Information and Software Technology Journal*, Vol. 36, No 7, pp 397-404,

Gorton I., Jelly, I.E. and Chan, T.S. (1994) Engineering High Quality Parallel Software Using *PARSE in Proc CONPAR,* September 1994, Linz, Austria, Springer Verlag LNCS, pp 381-392

Birkinshaw, C.I. and Croll, P.R. (1995) Modelling the Client-Server Behaviour of Parallel Real-Time Systems Using Petri Nets" *in Proc HICSS-28, (Hawaiian International Conference on System Sciences),* January 1995, Maui, Hawaii, USA, IEEE Computer Society Press Vol. II, pp 339-348

Gorton, I., Jelly, I.E., Croll, P.R. and Nixon, P. (1995) Directions in Software Engineering for Parallel *Systems in Proc HICSS-28 (Hawaiian International Conference on System Sciences)* January 1995, Maui, Hawaii, USA, IEEE Computer Society

Gorton, I., Jelly, I.E. and Gray, J P. (1995) Object Based Modelling of Parallel Programs *in IEEE Parallel and Distributed Technology Journal,* Vol. 3, No. 2, 1995, IEEE Computer Society Press

Lloyd D.W., Jelly, I.E. and Cai, J. (1995) Evaluation of PARSE for High Level Codesign Specifications *in Proc ICRAM '95 (Int Conf on Recent Advances in Mechatronics),* August 1995, Istanbul, Turkey

Jelly, I.E., Croll, P.R., Birkinshaw, C.I. and Gorton, I. (1995) Client-Server Behaviour Modelling in PARSE in *Proc Euromicro '95 Conference*, Como, Italy, September 1995, IEEE Computer Society Press

Knowles, C., Jelly I.E. and Collingwood, P.C. (1995) Evaluation of Software Engineering Analysis Techniques for Parallel Software *in Proc Euromicro 95 Conference*, Como, Italy, September 1995, IEEE Computer Society Press

Jelly, I.E. and Gorton, I. (1995) Directions in CASE Technology for Parallel Software Development *in Transputer Communications Journal,* Vol. 3. No. 1, pp

Liu, A. and Gorton, I. (1996) Modelling Dynamic Distributed System Structures in PARSE *in Proc 4th European Workshop on Parallel and Distributed Processing*, January 1996, Braga, Portugal, IEEE Computer Society Press

Gorton, I., Jelly, I.E., Gray, J.P. and Chan, T.S. (1996) Reliable Parallel Software Construction using PARSE *to appear in Concurrency: Practice and Experience Journal*

Sadler, D.R., Lloyd, D.W. and Jelly, I.E. (1996) Object Based Hardware/Software Co-design *to appear in Proc IEEE Int Conference on Computers and Communications,* March 1996, Phoenix, USA, IEEE Computer Society Press

5.2 Workshop Proceedings

Workshop on "Software Engineering for Parallel Systems", Aachen, Germany, September 1993. Selected papers published in special edition of Information and Software Technology Journal, Vol. 36, No. 7, 1994. Guest editors, I.E. Jelly, I. Gorton and J. P. Gray

Workshop on "CASE Technology for Parallel Systems Development", Como, Italy, September 1994. Selected papers in special edition of Transputer Communication Journal, Vol. 3, No. 1 (1995). Guest editors, I. E. Jelly and I. Gorton

Mini-track on "Software Engineering for Parallel Systems", in Proc HICSS-28 (Hawaiian International Conference on System Sciences), January 1995, Maui, Hawaii, USA, IEEE Computer Society 1995. Co-ordinators: I. Gorton, I. E. Jelly, P. R. Croll and P. Nixon

25

The AL++ project: object-oriented parallel programming on multicomputers

M. Di Santo, F. Frattolillo, W. Russo and E. Zimeo[*]
University of Salerno - [*]University of Napoli, Italy
Fax: +39 89 964574 - E-mail: disanto@dia.unisa.it

Abstract

AL++ is a software system which combines high-level object-oriented facilities with the simplicity, flexibility and power of the Actor computational model. AL++ lets programmers develop C++ parallel applications and run them on multicomputer platforms.

Keywords

Parallelism, Multicomputers, Actors, C++, Object-oriented parallel programming

1 PROJECT AIMS

Object-oriented programming models provide an attractive base for developing parallel programming systems. In fact, they promote the effective application of modern software engineering techniques, which have already proven to be successful in developing complex and large-scale sequential applications. Moreover, thanks to the dynamic creation and reconfiguration of objects, they also support applications whose computational structures can not be statically determined and facilitate decisions about object placement and migration, by aggregating data and code into single semantic units. In short, object-oriented parallel models seem to offer the expressiveness and the efficiency which are needed to effectively harness the computational power of modern, distributed-memory multicomputers.

Among the object-based models of parallel computation, *Actors* (Agha, 1986) is the best known. It can be classified as a partly abstract model based on process nets (Skillicorn, 1993) which allows computations to be specified without restricting their form. The Actor model has recently become the basis for a number of parallel object-oriented programming languages, such as ABCL (Yonezawa, 1990), CA (Chen, 1993) and HAL (Agha, 1992), even though it still has to establish itself as a practical tool for the development of parallel software. This is due to the difficulties encountered in turning the model into a truly general-purpose, object-oriented parallel programming language, to the scarcity of efficient implementations and to the limited experience for significant applications. In conclusion, the object-oriented approach, particularly if based on the Actor model, is well-suited for structuring parallel activities, but many further research and implementation efforts are needed in order to provide parallel programmers with elegant language ideas efficiently implemented on existing hardware.

These considerations have motivated the AL++ project which began in 1990 as one of the research proposals to be developed within the national project “Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, sottoprogetto Architetture Parallele”, sponsored by the *National Research Council* (CNR) of Italy. Part of the research activities were also developed within the project “Architetture Convenzionali e Non Convenzionali per Sistemi Distribuiti” sponsored by the *MURST* (Ministero dell’Universit e della Ricerca Scientifica e Tecnologica).

The AL++ project aims at developing a programming system characterized by the following requirements: (a) to provide programmers with elegant and simple mechanisms to develop object-oriented parallel applications on distributed-memory architectures; (b) to enable application code to be independent of underlying hardware/software platforms; (c) to achieve a modular implementation of the programming system so that it can be ported on new hardware with a reasonable effort.

2 KEY RESEARCH ISSUES AND ACHIEVEMENTS

Since the design and implementation of a new language is an expensive activity, the simpler approach of embedding Actor concepts and primitives into a widespread sequential programming language has been followed. In particular, the main achievement of the project has been *AL++*, a semantic extension of C++, implemented through a class library which provides an object-oriented interface for actor programming. The choice of C++ has been motivated by its availability and popularity with programmers. Another motivation is that C++ is efficiently implemented with a minimum of run-time support on all the architectures of major interest.

2.1 The actor model

Actors are objects which manifest a pure reactive nature and interact with other actors only via *message passing*. They unify both data and code in local states, called *behaviors*, and are dynamically created and referred through system-wide identifiers, called *mail addresses*

The communication mechanism is point-to-point, asynchronous and one-directional. Because mail addresses may be transmitted via messages, the actor-net which shows the potential flow of information may dynamically change. Messages are guaranteed to be delivered to their destinations, but transmission order is not necessarily preserved at delivery. Incoming messages are buffered into unbounded queues associated to receiving actors, before being serially processed. Functional interactions among actors are modeled with the use of *continuations*; that is an actor, instead of returning a result, sends it to a continuation actor that it knows about.

The processing of a message triggers the execution of the actor *script*, the code in the behavior of the receiver. During this processing, new actors can be created, messages asynchronously sent and the current behavior substituted by a new one (*replacement behavior*). In practice, replacements implement local state changes which can span from simple updates in the values of state variables to radical changes in the set of state variables and in the script.

2 The AL++ interface

AL++ enables programmers to exploit software engineering techniques in modeling parallel applications. In fact, AL++ joins C++ object-oriented powerful facilities, such as data abstraction, multiple inheritance, overloading and dynamic binding, with the clear, simple and flexible mechanisms provided by the Actor model. Moreover, thanks to the support for automatic and dynamic resource management, programmers can design AL++ programs as *ideal* algorithms, without having to specify allocation strategies or other programming details which make them depending on specific hardware platforms and network topologies.

AL++ supports the *SPMD* (Single Program Multiple Data) computational model; therefore, each node in the system stores and runs the same program; data, on the contrary, is distributed among all the nodes. The library makes available all the basic abstractions and primitives of the Actor model. Messages and behaviors are dynamically created instances of user classes respectively derived by the library classes *Message* and *Behavior*, while mail addresses are instances of the library class *MailAddress*. Actors are dynamically created by invoking the member function *MailAddress::create*. The behavior of the new actor can be specified at the creation time or later, by invoking the member function *MailAddress::init*. In the latter case it is possible to create actors whose behaviors mutually refer.

The *Message* class defines all the communication and message management primitives as its member functions. In particular, *Message::send* sends a message to a target actor, while *Message::request* associates to the sent message the identity of a *continuation* actor, which will be used as the implicit destination of the result when the target actor executes *Message::reply*.

Each user class derived from *Behavior* must include the local data as its data members and define the pure virtual member function *script*, which accepts the message to be processed as an argument. In many cases the script selects, on the basis of the tag associated to the message and returned by *Message::type*, the appropriate method and invokes it. *Behavior::become* permits to specify the actor replacement behavior, while *Behavior::self* returns the mail address of the current actor.

AL++ enables to control the dynamic placement of actors in two ways: (1) automatically, by employing one of the dynamic load balancing strategies integrated into the runtime support: *random*, *ACWN* (Shu, 1989) and *PWFA* (Di Santo, 1995, in preparation); (2) in a programmed way, by utilizing some primitives which allow both to specify the node on which an actor is to be created and to *migrate* actors according to the computation load at run-time. Moreover, immutable actors may be *duplicated* and "garbage" actors explicitly *deallocated*.

2.3 Implementation issues

The AL++ interface is built on top of a runtime support, called *ASK* (*Actor System Kernel*), which has been designed so as to fully exploit the power of the underlying hardware, and to be flexible enough to represent a stable basis for further enhancements. A working prototype of the kernel has been developed for Transputer networks.

To make the kernel portable to different hardware/software platforms and independent of network characteristics, it is built on top of a low-level interface which consists of two components: an *abstract node environment*, providing each node with facilities for running concurrent threads which interact through some shared-memory mechanism (semaphores or equivalent), and an *abstract network environment*, providing node-to-node asynchronous communication primitives and taking charge of performing routing between non-adjacent nodes and of buffering incoming messages.

An instance of the kernel, consisting of a few threads implementing system processes, is present on each node. One of these threads is the *scheduler* which cyclically schedules a local actor and processes messages in its mail queue; it is worth noting that the processing of a message can not be suspended and, therefore, once started, proceeds till its completion. Another thread is the *server* which carries out the remote requests as though they were issued locally.

Mail addresses are represented with global identifiers generated according to a completely distributed scheme that does not introduce overhead. The identifiers are then translated into physical addresses by a *lookup table* that returns either a local memory address, or a node identifier, according to the physical allocation of actors. In the latter case, a system message is sent to that node, and a new access to the lookup table is performed upon arrival.

Migration can be implemented quite cheaply in an actor based system. In ASK all the steps needed are fully asynchronous and so, while the actor migration proceeds on a node, other activities allocated on the same node have not to wait, but they are allowed to do useful work. Migration times are therefore masked by the resulting parallel execution of system threads, and they only affect the response time of the messages in the queue of the migrating actor. The migration procedure has been adopted as a basis to implement the *remote creation* primitive. In fact, an actor is always locally created and only then asynchronously migrated to its remote destination. This mechanism permits to minimize the time spent for an actor creation and to maximize the locality of data references in the first phase of actor existence.

2.4 Performances

The prototype implementation of ASK has been developed in the 3L Parallel C programming environment, and runs on a network of sixteen T800, clocked at 20 MHz, with links at 20 Mbits/s. Two versions of the network environment (NE) are available, respectively for ring-connected and 2D-torus networks.

Table 1 shows execution times of the four basic AL++ primitives (creation of a new actor, assignment of an initial behavior to a new actor, sending of a void message and replacement of the current behavior) in the case of purely local execution.

Table 1 Local execution of some AL++ primitives (μs)

create	*init*	*send*	*become*
39	54	223	65

Table 2 shows execution times of the *send* primitive as a function of the distance in *hops* of the target node. The table also reports the overall amount of time spent in the NE. The execution time of a remote *create* is constantly equal to 71 μs, in that ASK always performs a local creation asynchronously followed by a migration of the actor.

Table 2 Remote execution of the *send* primitive (μs)

	1 hop	*2 hops*	*3 hops*	*4 hops*	*5 hops*	*6 hops*	*7 hops*	*8 hops*
send(NE)	382(113)	466(177)	535(240)	604(305)	690(368)	771(432)	854(496)	920(583)

2.5 Bibliography

In the following we provide a list of the AL++ key publications written in English:

Arcelli F., De Santo M., Di Santo M. and Picariello A. (1993) Computer Vision Applications Experience with Actors, *PARLE '93*, 14-18 June 1993, Munich (Germany), *LNCS 694*, Springer-Verlag, Berlin (Germany).

Di Santo M. and Iannello G. (1990) ASK: A Kernel for Programming Actor Systems, *Procs. of the 1990 ACM SigSmall/PC Symposium on Small Systems*, ACM Press.

Di Santo M. and Iannello G. (1991) Implementing actor-based primitives on distributed memory architectures, *Procs. ECOOP-OOPSLA Workshop on Object-Based Concurrent Programming*, 21-22 Oct. 1990, Ottawa (Canada), *OOPS Messenger* 2(2), ACM Press.

Di Santo M. and Iannello G. (1992) Implementation of dynamic languages on multicomputer architectures, in *Parallel Computing: Problems, Methods and Applications* (eds. Messina P. and Murli A.), Elsevier, Amsterdam (Nederland), selection of papers presented at the *Conference on Parallel Computing: Achievements, Problems and Prospects*, 3-7 June 1990, Capri (Italy).

Di Santo M., Iannello G. and Russo W. (1992) ASK: a Transputer implementation of the Actor model, *Int'l Conf. on Parallel Computing and Transputers Applications*, 21-25 Sept. 1992, Barcelona (Spain), IOS Press, Amsterdam (Nederland).

Di Santo M., Frattolillo F. and Iannello G. (1992) *Actor System Kernel (ASK) 4.0. Introduction and User Guide*. Tech. Rep. n. 3/108, CNR Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo.

Di Santo M., Frattolillo F. and Iannello G. (1994) *Run-time support for highly parallel algorithms on multicomputer architectures*, Tech. Rep. n. 3/139, CNR Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo.

Di Santo M., Frattolillo F. and Iannello G. (1995) Experiences in Dynamic Placement of Actors on Multicomputer Systems, *Proceedings Euromicro Workshop on Parallel and Distributed Processing*, San Remo 25-27 Jan. 1995 (Italy), IEEE Computer Society Press.

Di Santo M. and Iannello G. (1995) Actor Models, in *General Purpose Parallel Computers: Architectures, Programming Environments and Tools* (eds. Balbo G. and Vanneschi M.), Edizioni ETS, Pisa (Italy).

Di Santo M., Frattolillo F., Russo W. and Zimeo E. (1995) *A Dynamic Load Balancing for Object-Based Computations on Multicomputers*, in preparation.

3 FUTURE DIRECTIONS

The AL++ project is still alive and we want to utilize the experiences accumulated since its start in order to globally redesign both its interface and implementation. Precisely, at the interface level, we will proceed to substitute C++ with the new object-oriented language *Java* (Gosling, 1995) which will offer the advantages of being, according to its authors: (i) simple and familiar; (ii) architecture neutral, portable and robust; (iii) interpreted, dynamic, secure and multi-threaded; (iv) efficient and equipped with extensive and well developed class libraries. Moreover, we will complete the programming interface with mechanisms for expressing *local synchronization constraints*, which permit to delay the processing of messages until they are "serviceable", and *grouping of actors*, which permit to express data parallelism, to support broadcast communication and to implement distributed objects.
On the other hand, at the implementation level, we will move our environment to a *network of workstations* equipped with *PVM* (Geist, 1992), which nowadays have proven to offer viable and cost-effective platforms for parallel computing in many application domains. Moreover, we will design and implement new mechanisms for explicit and automatic resource management at runtime: among these we will include new algorithms for dynamic placement and distributed garbage collection of actors.

4 ACKNOWLEDGMENTS

The AL++ project has been partially supported by CNR, under funds of "Progetto Finalizzato

Sistemi Informatici e Calcolo Parallelo", and by MURST, under funds 60% and 40% "Architetture Convenzionali e Non Convenzionali per Sistemi Distribuiti".

In addition to the authors, Giulio Iannello of the University of Napoli "Federico II" has contributed to the project.

We gratefully acknowledge Gul Agha of the University of Illinois at Urbana-Champaign for inspiring our work and providing insights into the Actor model and its implementation.

5 REFERENCES

Agha G. (1986) *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press.

Agha G. and Houck C. (1992) HAL: A High-level Actor Language and Its Distributed Implementation, *Proceedings of the 21st International Conference on Parallel Processing* (ICPP '92), Aug. 1992, St. Charles (IL - USA).

Chien A. (1993) *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. The MIT Press.

Geist G. A. and Sunderam V. S. (1992) Network-Based Concurrent Computing on the PVM System, *Concurrency: Practice and Experience*, 4(4).

Gosling J. and McGilton H. (1995) *The Java Language Environment: a white paper*, available at http://java.sun.com

Shu W. and Kalè L. V. (1989) *Dynamic scheduling of medium-grained processes on multicomputers*, Tech. Rep., Dep. of Computer Science, Univ. of Illinois at Urbana-Champaign.

Skillicorn D. B. (1993) Models for parallel computation, in *Advanced workshop on Programming tools for parallel machines*, 21-25 June 1993, Otranto (Italy).

Yonezawa A. (ed.) (1990) *ABCL: An Object-Oriented Concurrent System*. The MIT Press.

6 BIOGRAPHIES

Michele Di Santo is a professor of computer engineering at the University of Salerno, Italy. He received the degree in electronic engineering, cum laude, from the University of Napoli and worked at the University of Napoli and the University of Calabria. His scientific interests include programming languages and environments for parallel and distributed systems. He is a member of ACM and IEEE Computer Society.

Franco Frattolillo is a faculty member at the "Dipartimento di Ingegneria dell'Informazione e Ingegneria Elettrica" of the University of Salerno, Italy. He received the degree in electronic engineering, cum laude, from the University of Napoli. His research interests include parallel and distributed architectures and programming environments for parallelism.

Wilma Russo is an associate professor of computer engineering at the University of Salerno, Italy. She received the degree in physics, cum laude, from the University of Napoli and worked at the University of Calabria. Her scientific interests include programming languages and environments for parallel and distributed systems.

Eugenio Zimeo holds a scholarship from CNR at the University of Napoli, Italy. He received the degree in electronic engineering, cum laude, from the University of Salerno. His research interests include parallel and distributed architectures and programming environments for parallelism.

26
The Basel Tool Suite for Parallel Processing

H. Burkhart, N. Fang, R. Frank, G. Hächler, W. Kuhn, and G. Prétôt
Universität Basel
Institut für Informatik, Mittlere Strasse 142, CH-4056 Basel, Switzerland.
Telephone: +41-61 321 99 67 Fax: +41-61 321 99 15
email: **burkhart@ifi.unibas.ch**

Abstract

The Basel approach is characterized by a coordinated set of subprojects that use the same basic terminology but target for different goals. Emphasis is put on software engineering aspects such as increased programmability, portability, and interoperability. This integrated approach has benefits because different user groups (application programmers, performance analysts, students) can interact and profit from synergies by using common system elements. The Basel Tool Suite (developed at PUB*) currently consists of

- ALWAN, the language used for writing algorithmic skeletons, offering reuse-in-the-small constructs, and its compiler which produces portable source code skeletons for different target systems and programming languages.
- PEMPI, a library-based environment supporting structured parallel programming for the MPI message passing standard.
- ALPSTONE, a methodology and environment to make performance prediction and algorithmic benchmarking on different target architectures.

Prototypes for all components exist and are used in courses at the University. Public domain versions of this software are envisaged; potential users should contact the group.

Keywords

software engineering for parallel and distributed systems; structured parallel programming; mixed language programming; portability; MPI support; performance prediction; algorithmic benchmarking; parallelism courseware.

*Parallel laboratory, University of Basel

1 ALWAN : A PARALLEL COORDINATION LANGUAGE

ALWAN is a parallel language and programming environment developed at PUB. The design goals of ALWAN are to increase the programmability of parallel applications, enable performance portability, support the reuse of software components, and mixed-language programming. Parallel programs consist of (sequential) calculation and (parallel) coordination parts. To address the major difficulties in parallel programming, the ALWAN language provides high level constructs for the description of parallel coordination aspects such as data partitioning and distribution, process topology management and communication aspects. As ALWAN is intended to specify only the coordination of an algorithm, it provides an interface to other, widely used, sequential languages, such as C and FORTRAN. Coordination skeletons and sequential building blocks are processed by the programming environment (ALWAN compiler and support libraries) which can automatically generate programs for various parallel architectures.

Approach

Program development within the ALWAN environment is outlined in the figure below to which the Roman numerals refer.

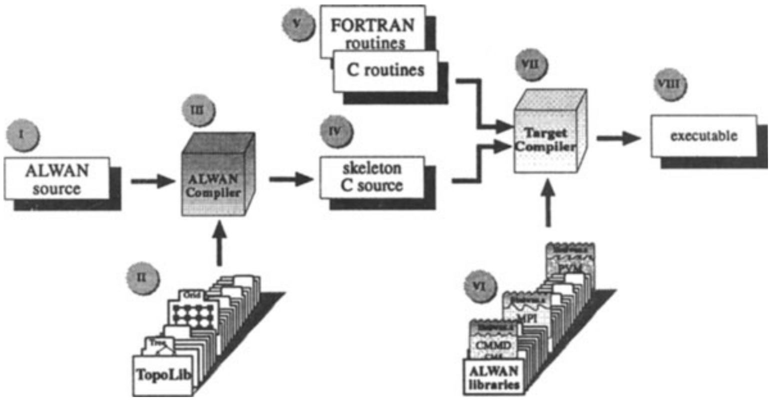


Figure 1 system overview

A coordination description (I) written in ALWAN is transformed into a source code skeleton (IV) using the ALWAN compiler (III). Predefined ALWAN modules, where frequently used topologies or routines are collected, may be imported and re-used (II). Procedures declared as EXTERNAL define the interface to code written in other (sequential) languages (V). The high-level parallel coordination constructs are translated into ALWAN library (VI) calls with appropriate parameters. This library is implemented for various machines, interfacing to the virtual machine layers available on the given platform. Finally, all code parts are compiled (VII) and linked to form an executable program (VIII). Porting to a different platform only requires a recompilation on the target machine, thus replacing the ALWAN library with the appropriate new one. Compilation of the different source parts (I, IV, and V) and linking to the appropriate libraries (II and VI) is handled by a shell script and generated make files.

Status and Future Work

The ALWAN compiler is implemented on various UNIX platforms. The full set of library routines has been implemented for PVM. To prove the feasibility of our approach, an intermediate version (no support of collective communication, inhomogeneous data, asynchronous communication, and run time checks) was implemented for PVM, MPI, CMMD, and NX and tested on CM5, SP1, Paragon and a workstation cluster containing NeXT and Sun workstations. The compiler supports mixed-language programming in that the external routines may be written in either C or FORTRAN 77.

A full ALWAN implementation for PVM and MPI will be available in the first quarter of 1996. Further implementations will follow and include support for virtual shared memory systems.

The ALWAN tool suite is currently used within different projects: ALWAN is used within the ALPSTONE performance prediction environment (described in section 3). A library of sample parallel algorithms was built to be used as teachware. This library contains matrix multiplication variants and stencil algorithms (both using a torus topology), bitonic sort using a hypercube topology, divide and conquer sort on a tree, gaussian elimination and transitive closure of graph (Warshall algorithm) both on a farm. Other members of the laboratory use ALWAN to solve application problems (CFD code; image processing for computer-aided surgery). Another project extends ALWAN to support the creation of parallel services for industry standard client server environments.

Contact Person	: Robert Frank and Guido Hächler ({frank|haechler}@ifi.unibas.ch)
Home Page	: http://www.ifi.unibas.ch/~alwan
Reference	: Burkhart, H., Frank, R. and Hächler, G. (1996) Structured Parallel Programming: How Informatics Can Help Overcome the Software Dilemma. To appear in *Scientific Programming, 1996.*

2 PEMPI : PROGRAMMING ENVIRONMENT FOR THE MESSAGE PASSING INTERFACE

The PEMPI project helps the programmer in writing message passing programs by using higher abstraction functions and supporting tools. It aims to achieve three goals in a unified approach: *obtain portability by employing the MPI standard, achieve performance through the machine best-fit implementation, and increase programmability by exploiting higher abstractions and taking advantage of supporting tools.*
As a design feature, PEMPI allows the application programmer to work within the context of the higher-level functions to solve regular problems but also to jump into the system level, i.e., MPI level, to solve irregular or performance-critical problems if necessary.

Architecture and Functionality

PEMPI is a two-layer architecture: Kernel PEMPI (KPEMPI) and Outer PEMPI (OPEMPI), as Figure 2 shows. Kernel PEMPI contains the higher abstraction library based on MPI, which can be further divided into three modules: (1) *Process Manage-*

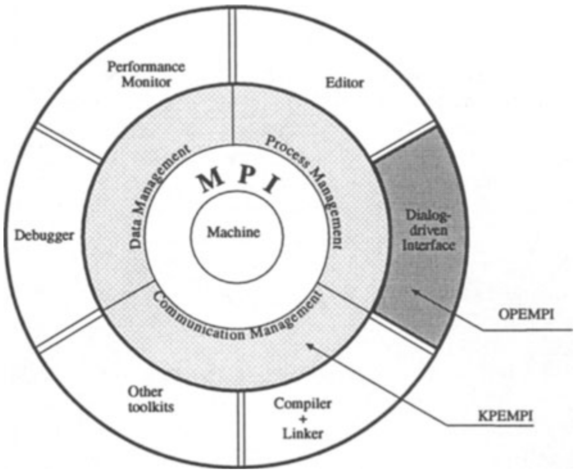


Figure 2 PEMPI Two-layer Structure

ment Module: containing functions to build process topologies such as a tree or hypercube besides the topologies supported in MPI, and functions to access topology information; (2) *Data Management Module*: containing functions to input/output and distribute/collect data according to the layout of the data partition which is specified by the programmer as well as functions structuring commonly-used data components before they are sent/received; (3) *Communication Management Module*: containing some topology-specific communication routines.

Auxiliary functions such as index conversion (global to local and reverse) of distributed data are also supported.

Outer PEMPI consists of a collection of tools: (1) a dialog-driven programming interface: a template generator with GUI is used to generate code related to process topology creation, data layout specification, data I/O and distribution/collection etc; (2) a text editor; (3) target system utilities: including a compiler, linker and debugger; (4) toolkits: e.g. for automatically converting C structs and unions to MPI Datatypes.

Status and Future Work

KPEMPI has been implemented and has partially been tested and ported to the IBM SP-2 at Argonne National Laboratory and the Fujitsu AP1000 at Imperial College of Science, Technology and Medicine/Fujitsu Parallel Computing Research Center, University of London. The prototype of OPEMPI is still under construction using the NEXTSTEP utilities Project Builder and Interface Builder.

Future work includes: (1) implementing the full OPEMPI prototype using NEXTSTEP; (2) porting KPEMPI to other parallel machines; (3) implementing sample parallel algorithms in PEMPI and also real-world applications such as CFD, image processing, etc; (4) porting OPEMPI to other user interfaces, e.g., OSF/Motif; (5) detailed analysis of performance loss caused by abstraction on different parallel machines and optimization of the code.

Contact Person	: Niandong Fang (fang@ifi.unibas.ch)
Home Page	: http://www.ifi.unibas.ch/~pempi
Reference	: Fang, N. and Burkhart, H. (1994) PEMPI—From MPI Standard to Programming Environment. *Proceedings of Scalable Parallel Libraries Conference II (SPLC'94)*, 31-38, Mississippi.

3 ALPSTONE : PREDICTION INCREASES PRODUCTIVITY

Beside correctness, portability, and reusability, efficiency is one of the most important questions to be answered when developing software for a parallel system. However, developing a program, running it and finding out that it is too slow is a bad approach. We need an early estimation of a program's performance to increase the programmer's productivity.

ALPSTONE's methodology guides a skeleton programmer in going from a formal description of an algorithm to a finished program from the point of view of performance studies and supplies an experimental algorithm performance test bed. Techniques included are benchmarking and performance prediction. Fig.3 shows an overview of the architecture.

Approach and architecture

Early in the software engineering phases, a parallel program is modeled by a macroscopic abstraction (I) containing program properties such as process topology, execution structure, data descriptions, I/O behaviour, and interaction specifications. This information may be refined during the software engineering process where necessary.

The *Extractor* (II) now derives a time model and may (later) generate a rudimentary, instrumented skeleton program (e.g. ALWAN or PEMPI). The user has to supply additional information, e.g. code for a local computation or refinements of the skeleton.

The generated time model predicts the program performance in terms of the abstraction. The *Estimator* (III) supports the performance prediction steps through the whole software engineering cycle by calculating the model's runtime and supporting its refinement. The estimator detects idle times or allows the overlapping of computation and interaction. Other information is available as well, such as the amount of certain operations or transferred data. Statistical models for runtimes and the modelling of an inhomogeneous system are supported. In order to be as accurate as possible, a library of performance data representing topology creation, data distribution times, etc is mandatory (VI).

If the prediction promises good performance, program implementation and translation language will follow (IV). Otherwise, modifications will be done. This early detection of performance faults is important as this is where programmer productivity can be increased.

After having run an instrumented program, an *Analyzer* (V) stores measurements (for reuse in other estimations) in the performance database. Of course, comparing the predicted and measured runtime is of interest.

The ALPSTONE *benchmark suite* (VI) supplies the estimator with data from selected benchmarks. We include some well known codes, but define some new benchmarks as well.

The approach is hierarchical because the building blocks of a parallel program will often be found in complex routines composed of actions from the lower levels, in that way detailed measurements of building "blocks" may refine first estimations based on the "base" level. The suite is vertically structured using the basic properties modeled in the performance skeletons.

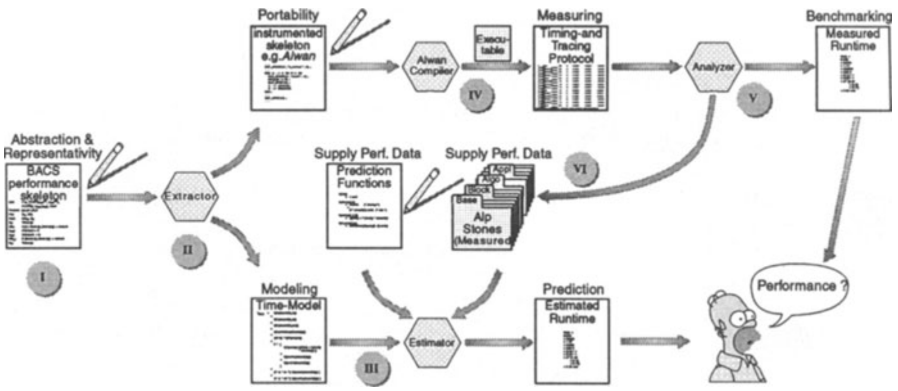


Figure 3 An overview of the ALPSTONE environment

Status and future work

First case studies showed that it is worth integrating performance studies in the software engineering cycle while formulating an outline of a program. Today, most of the macroscopic specification and test cases are defined. The ALPSTONE benchmark suite is specified in three layers and most benchmarks are implemented on several machines. The *Extractor* is currently realized using a compiler toolbox. The *Estimator*, defined as a layered system, and the *Analyzer* are running in the Mathematica environment on a subset of the generated models. Finally, an instrumentation library has been written in C.

We will finish the implementation, port our benchmarks to different systems and predict more skeletons in the next steps. Directions where ALPSTONE can be used or extended are: a suitability study of algorithms on different architectures, a support for load balancing strategies, the improvement of the estimation for network resources, or studying the behaviour of implementation strategies without the need of accessing a parallel system.

Contact Person	: Walter Kuhn (kuhn@ifi.unibas.ch)
Home Page	: http://www.ifi.unibas.ch/~alpstone
Reference	: Kuhn, W. and Burkhart,H. (1995) The ALPSTONE project: An Overview of a Performance Modeling Environment. Proceedings of the Conference on High Performance Computing (HiPC'95), New Delhi, 1995.

PART THREE

Demonstrations

27

Development Framework for real-time control system design

J. M. Bass[†], *A. R. Browne*[†], *M. S. Hajji*[†], *P. R. Croll*[‡] *and P. J. Fleming*[†]
[†]*Dept. of Automatic Control and Systems Engineering,*
[‡]*Dept. of Computer Science, University of Sheffield, Mappin Street, Sheffield, S1 3JD, UK. Tel. + 44 (0)114 282 5236,*
Fax. + 44 (0)114 273 1729, E-mail: J.Bass@sheffield.ac.uk

Abstract
The Development Framework provides a highly automated translation from a specification to a parallel implementation. The specification is in a popular graphical control engineering notation, typically representing a system with stringent dependability requirements and hard real-time constraints. An interface has been constructed between the Development Framework and the dependability modelling tool, SURF-2. The demonstration will illustrate the Development Framework design approach using a primary flight control Case Study. The example application consists of a three channel autopilot and airframe model. Dependability models of competing autopilot architectures will be contrasted in the demonstration.

Keywords
Computer-aided control system design, dependability modelling, computer-aided software engineering, real-time systems, distributed systems.

1 INTRODUCTION

The Development Framework, an environment to support the specification, design and implementation of real-time distributed computer control systems is described here. It is argued that both good design practice and fault-tolerance are required to ensure that stringent reliability targets are met. Distributed computer control systems have the advantage that redundant processing elements are available for use to provide fault-tolerance.
The Framework, provides support for three phases in the development of the system under design. The Specification Phase, described in Section 2, allows the designer to specify, analyse and simulate the control system under development. The Development Framework includes tools that automatically translate the control engineering representation into a software engineering representation. The Software Design Phase uses a software engineering notation, described in Section 3, to enable analysis and refinement of the system under development. One type of analysis available to the developer is the generation of stochastic Petri net dependability models, described in Section 3-1. Further Development Framework tools translate the software engineering representation into source code that can be compiled into executable code for a network of processors. The resulting parallel implementation is discussed in Section 4.

The demonstration will use a Case Study to illustrate the Development Framework approach. The Case Study is not described in detail here, due to lack of space, but is introduced in Section 5. Conclusions are provided in Section 6. Further information regarding the Development Framework can be found in (Browne, 1994) and (Bass, 1994).

The Development Framework addresses a similar problem area to the ControlH/MetaH design environment (Vestal, 1994). In common with the Framework, the ControlH/MetaH tools use an application-specific graphical specification notation and an intermediate software engineering notation. However, the Framework integrates commercially available tools using translators, while ControlH/MetaH is implemented entirely using purpose built tools. Further, the ControlH/MetaH environment does not provide facilities for dependability modelling. Detailed dependability modelling, without the benefits of system specification, design and implementation support, can be performed using Markov chains or stochastic activity networks. The SAVE environment uses a textual system description to provide dependability measures (Blum, 1993). In contrast, the UltraSAN environment provides a graphical interface based on stochastic activity networks (Sanders, 1993). The dependability modelling tool selected for this work, SURF-2, is Markov-based using stochastic Petri nets.

The Development Framework approach encourages the designer to concentrate on the control engineering design aspects of the proposed system. This is achieved by providing a highly automated path from a control engineering specification to a distributed system implementation. Figure 1 shows the three phases supported by the Framework and the main benefits provided in each phase. The Development Framework provides an open architecture to encourage the designer to intervene at appropriate stages of the design lifecycle for the purposes of optimisation.

2 SPECIFICATION PHASE

The specification of software with the use of diagrams is seen as one of the main advantages of CASE systems. It is generally recognised that diagrams allow the representation of system structure in a much more accessible and natural form than written language or mathematics. Graphical notations have been developed that are appropriate for the specification of control systems and are used within the Development Framework. Therefore a control engineer should readily be able to understand the specification of a control system in such a notation. This would not generally be true if the design of the control system was in, for example, a

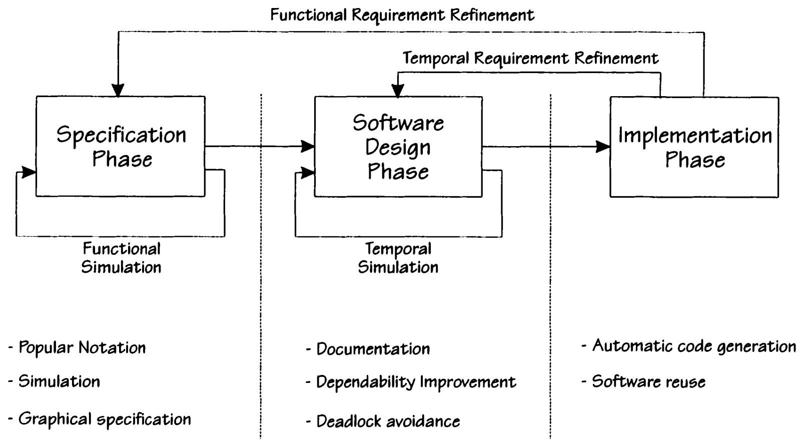


Figure 1 Development Framework overview.

software engineering notation. Simulink was selected for the specification of real-time control systems in the Development Framework because it: accommodates both continuous and discrete elements, and supports the hierarchical decomposition of diagrams enabling representation of complex control systems. Simulink supports modelling and simulation during control law design and is also used to provide a well documented mechanism for the specification of control systems. Simulation enables verification that the system meets requirements prior to implementation. The notation used by Simulink, in common with similar notations, was not designed to represent many of the features central to parallel and distributed systems, however. The Software Design Phase is, therefore, implemented to enable deadlock analysis, mapping and, if required, dependability analysis and the introduction of fault-tolerant mechanisms.

3 SOFTWARE DESIGN PHASE

The most novel and powerful feature of the Development Framework is the automatic translation of specifications, using an application-specific notation, into designs, using a generalised software engineering notation. An equivalent dataflow diagram is created for each Simulink diagram within a model, and a data structure diagram is created for every connection between blocks in each Simulink diagram. All functional blocks within the Simulink diagram (gains and transfer functions, for example) are converted into equivalent process symbols. Each Simulink inport/outport symbol is converted into an off-page connector, allowing processes and their decompositions to be linked. Thus, a complete description of the application system under design is maintained in the CASE tool. This complete description is required to allow the analysis, implementation and documentation of the proposed design.

The Framework draws on the CSP message passing paradigm (Hoare, 1985). The message-passing approach of CSP provides an elegant platform for the development of such distributed systems. Dataflow diagrams are used to model concurrent processes and message-passing channels. CSP-based processes and communication channels are, thus, conveniently modelled using CASE tools. The CASE tool environment, Software through Pictures (StP), was adopted for the Development Framework project because it: supports the well documented and widely known Yourdon methodology with Hatley/Pirbhai real-time extensions (Hatley, 1987); enables the generation and manipulation of diagrams with minimal user intervention; and has a flexible and extendible storage structure for specific information

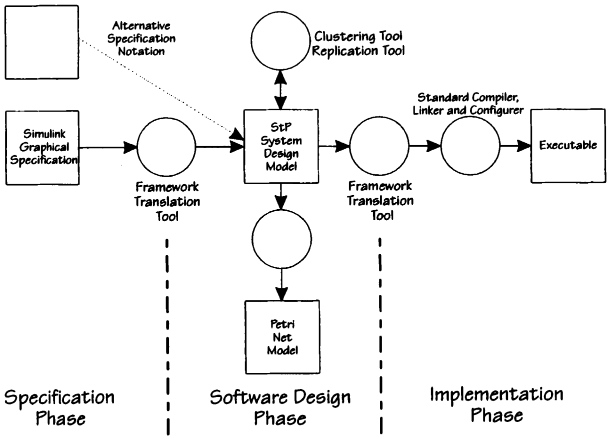

Alternative Specification Notation
Clustering Tool
Replication Tool
Standard Compiler, Linker and Configurer
Simulink Graphical Specification
StP System Design Model
Executable
Framework Translation Tool
Framework Translation Tool
Petri Net Model
Specification Phase
Software Design Phase
Implementation Phase

Figure 2 Development Framework tools.

about each object (diagram, process, data flow etc.) within the system.

Tools to perform replication of selected processes, generation of hierarchical coloured Petri nets and to cluster processes have been implemented. These allow analysis or perform optimisations on the distributed system under development in the software engineering domain. These optimisations can be performed with minimal intervention by the user. An approach to generating dependability models of the system under development is described below.

3-1 Stochastic Petri net tool

Generalised stochastic Petri nets enable the evaluation of system safety and reliability measures. The SURF-2 environment performs model processing based on graphical Petri net (or Markov chain) representations (Béounes, 1993). The SURF-2 Gateway, shown in Figure 3, supports automated generation of Petri nets from external software tools. The Framework stochastic Petri net tool analyses the system under development and translates the dataflow representation into a textual Petri net notation (Bass, 1995). Dependability models of selected fault-tolerant mechanisms are currently supported. Figure 4 shows typical translations for recovery block and n-version systems. The dependability models can be used to perform sensitivity analysis or contrast competing system architectures.

4 IMPLEMENTATION PHASE

A formalism is required in order to generate code from dataflow diagrams. Without this formalism there is no way of expressing the control of processes or the synchronisation of communications between them. The formalism represents each non-decomposed process symbol in the dataflow diagrams for a system as a separate process in the implementation. All these processes execute iteratively. In each iteration, the process: receives data from all input data flows; executes the functional code (transfer function or gain, for example); and sends data to all the output data flows. If a process has no input data flows the process waits for a signal from the process manager before executing the functional code. The process manager is a separate task responsible for the correct real-time operation of all the processes on a processor.

The formalism used limits the prototype Framework to the specification, design and im-

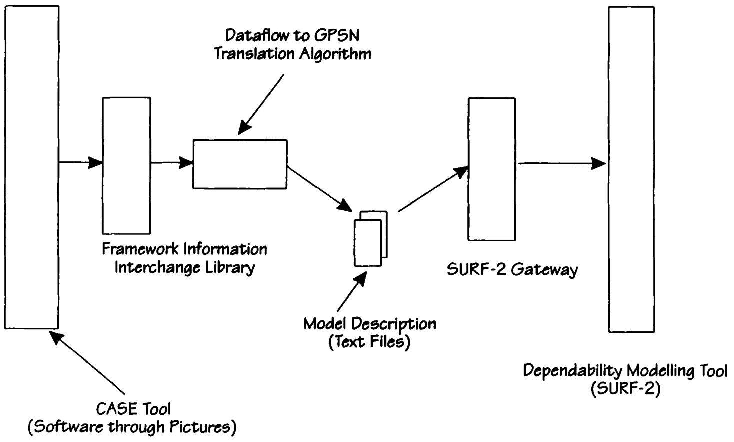

Dataflow to GPSN Translation Algorithm
Framework Information Interchange Library
Model Description (Text Files)
SURF-2 Gateway
Dependability Modelling Tool (SURF-2)
CASE Tool (Software through Pictures)

Figure 3 Development Framework to SURF-2 interface.

plementation of purely periodic systems. No concept of aperiodic tasks or events has yet been developed. All inter-process communication is strictly synchronous. The Framework currently produces source code in the language "C" for the Virtuoso real-time kernel executing on a network of Inmos Transputers. Transputers provide a convenient platform for the CSP model and have found numerous applications in real-time control (Irwin, 1992). The Virtuoso kernel includes a flexible, reconfigurable, synchronous message passing system and a rate-monotonic scheduler which makes it particularly suitable for the Framework.

The Framework code generator produces all the code required to compile, link and execute the system. For each process within the system two source code files are produced, a harness code file, and an application code file. The harness file contains code that manages inter-process communication and communication with the process manager. It is automatically generated to match the needs of the process. The application code file contains the code for the functional part of the process e.g. transfer function or gain. This code is an expansion of a template taken from a library of reusable source code modules. The development of such a library reduces both the implementation time, by automatically reusing existing code, and improves software reliability. The choice of a suitable library module for a process is performed automatically based on the number and type of input and output data flows and the type of routine (e.g. gain) that is required. This information is all stored within the CASE system when the control systems design is converted into data flow diagrams.

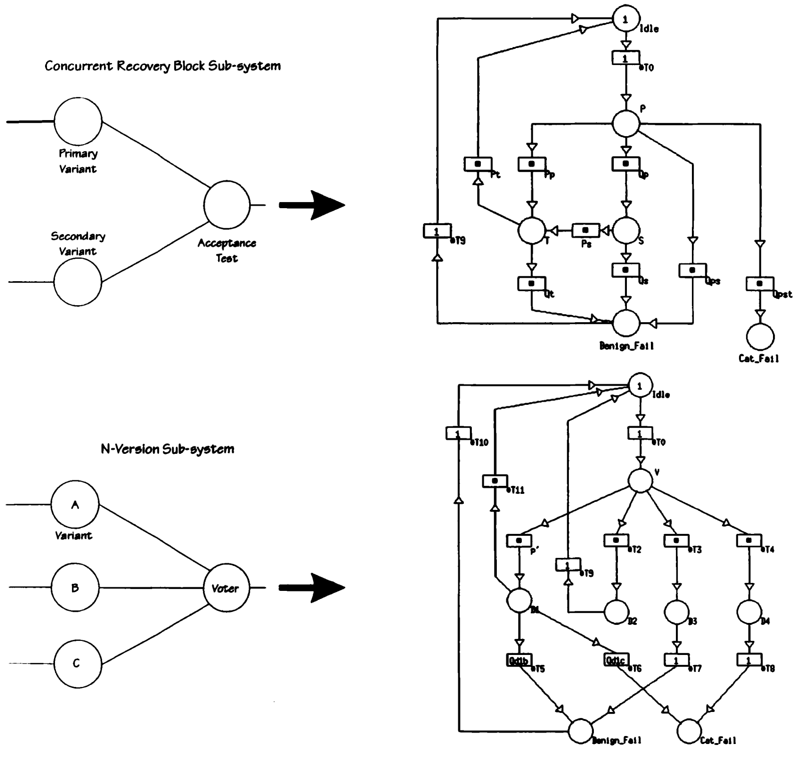


Figure 4 **Typical dataflow to stochastic Petri net translations.**

5 CASE STUDY

The software demonstration will use a primary flight control Case Study to illustrate the Development Framework design approach. The application consists of a generic three channel autopilot and airframe model.

6 CONCLUSIONS

The prototype Development Framework described here enables a highly automatic translation from an application-oriented system specification to an implementation executed on a parallel platform using a real-time kernel. In summary, the Framework approach offers a number of benefits. The system specification is in an application-oriented notation which can be simulated, to ensure correct functional behaviour, prior to implementation. Code re-use and automation of error-prone manual translations, reduce development time and increase confidence in implementation reliability. The open architecture provided by the Development Framework allows the addition of tools to address problems at different stages of the design lifecycle.

7 ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support of UK EPSRC (under grant number GR/K64310) and Intelligent Systems International, suppliers of the Virtuoso kernel.

8 REFERENCES

Bass, J. M., A. R. Browne, M. S. Hajji, D. G. Marriott, P. R. Croll and P. J. Fleming (1994), "Automating the Development of Distributed Control Software", IEEE Parallel and Distributed Technology, Vol. 2, No. 4, Winter 1994, pp. 9-19.

Bass, J. M., S. Metge, P. R. Croll and P. J. Fleming (1995), "Dependability Modelling in a Prototype Development Framework", IEEE 25th Ann. Int. Symp. on Fault-Tolerant Computing Systems, Pasadena, June 1995, pp. 131-6.

Béounes, C., et al (1993), "SURF-2: A program for Dependability Evaluation of Complex Hardware and Software Systems", Digest of Papers, IEEE 23rd Ann. Int. Symp. on Fault-Tolerant Computing Systems, Toulouse, June 1993, pp. 668-73.

Blum A. M. et al, (1993), "System Availability Estimator (SAVE) Language Reference and User's Manual", Research Report RA219S, IBM Research Division, T. J. Watson Research Centre, Yorktown Heights, N. J., June 1993.

Browne, A. R., J. M. Bass, P. R. Croll and P. J. Fleming (1994), "A Prototype Framework of Design Tools for Computer-Aided Control Engineering", Joint IEEE/IFAC Symp. on Computer-Aided Control System Design, 1994, pp. 369-74.

Hatley, D. J. and I. A. Pirbhai (1987), "Strategies for Real-Time System Specification", Dorset House Publishing Co. Inc.

Hoare, C. A. R. (1985), "Communicating Sequential Processes", Prentice-Hall.

Irwin, G. W. and P. J. Fleming (eds.) (1992), "Transputers in Real-Time Control", Research Studies Press.

Sanders, W. H. and W. D. Obal II (1993), "Dependability Evaluation using UltraSAN", Digest of Papers, IEEE 23rd Ann. Int. Symp. on Fault-Tolerant Computing Systems, Toulouse, June 1993, pp. 674-79.

Vestal C., (1994), "Integrating Control and Software Views in a CACE/CASE Toolset", IEEE/IFAC Joint Symp. on Computer-Aided Control System Design, Tuscon, Arizona, March 1994, pp. 353-58.

28

A Knowledge Based Approach to Parallel Software Engineering

P. Milligan, P. P. Sage, P. J. P. McMullan and P. H. Corr
The Queen's University of Belfast
Department of Computer Science
Belfast BT7 1NN
N. Ireland
Tel: +44 1232 245133 Extn. 4645
Fax: +44 1232 331232
E-mail: p.milligan@qub.ac.uk

Abstract

Advances in technology have resulted in the development of many different multiprocessor systems. Unfortunately these have not been accompanied by advances in portable, user-friendly program development environments. This paper overviews the prototype of a development and migration environment for parallel software engineering which incorporates the application of knowledge based techniques to the core topics of loop restructuring, code generation and code evaluation.

Keywords

CASE environments and support tools; application of AI (expert system) techniques; autoparallelisation.

1 INTRODUCTION

In the past few years there has been a dramatic increase in the number of different multiprocessor systems in the marketplace. This development has provided the user with an increased potential processing power and an even wider range of parallel machine architectures. However, this increase in power and choice has not been accompanied by an increase in flexible, portable, user-friendly program development environments. If the potential users of multiprocessor systems are to become actual users such environments must be provided.

It is fair to say that the majority of scientific and engineering users do not have the time or the desire to understand the intricacies of data dependence analysis, parallel program design and load balancing techniques for multiprocessor machines. For this potential user base a viable development environment must provide not only facilities for the development of new code but also an integrated toolset to ease the migration of their existing, mainly sequential, codes. The Fortport prototype (Milligan et al, 1992, Quill et al, 1995) described here is an attempt to provide such an integrated parallel software development and migration environment for Fortran programmers. Central to the ethos of the Fortport prototype is the belief that the user should have control over the extent of their involvement in the parallelisation process. It should be possible for novice users to be freed from the responsibility of detecting parallelisable sections of code and distributing the code over the available processors. Alternatively, experienced users should have the facility to interact with all phases of the parallelisation and distribution of the code. To enable a novice user to devolve responsibility entirely to the system implies that the system has sufficient expert knowledge to accomplish the task. This paper discusses how such expert knowledge has been provided within the Fortport prototype with particular emphasis on the migration of existing sequential code onto a multiprocessor architecture.

2 KNOWLEDGE ACQUISITION AND APPLICATION

One of the dominant problems associated with program development systems for parallel architectures has been the inability to completely automate the system. Several development environments exist, e.g. SUPERB (Zima et al, 1988) and PDE (Decker et al, 1993), but it is inevitable that they require some form of user interaction to assist with the process of code parallelisation. This interaction can take two forms, either the user annotates the program to indicate to a compiler that certain actions are required or the user interacts with the system during execution to choose transformations or data partitioning schemes.

The effect of the user interaction is to assist the development environment by providing user expertise. Hence it should be possible to develop expert systems to at best replace, or at worst supplement, this user interaction.

To investigate the viability of this approach two expert systems were proposed and developed for use within the FortPort prototype. One system would assist with the process of transformation selection and one with code generation and evaluation.

A common approach to the development of both expert systems was adopted. The approach was to hand code the required solutions and then analyse the decision making processes followed in the development of the code, i.e. a reverse engineering model was used. This model enabled the key steps in the two processes (parallelisation and generation/distribution) to be identified. In addition the key facts that trigger the various decision making steps could be identified. These key facts or characteristics again fall into two groups, namely loop characteristics and performance characteristics.

Subsequently the rules used in the expert systems were derived by combining the key steps identified in the reverse engineering phase with the relevant characteristics. In general the rules have the form:

```
define rule 'name'
   condition list
         =>   action.                                                    (1)
```

The condition list contains one or more expressions based on the characteristics which must be satisfied for the designated action to take place. When all of the conditions in a list are met the rule is fired. Examples of complete rules are given in a later section of the paper.

Future extensions to the knowledge acquisition phase will add characteristics derived from the application domain, prior and historical knowledge, i.e. decisions made in the past that may be applicable again. In other words a long term goal of the system will be to provide a learning environment.

3 THE PROTOTYPE SYSTEM

3.1 Input Handling and Graph Construction

The input handler generates a representation of the user program in the form of a graph. The graph is formed from a hierarchy of nodes. A detailed description of the graph nodes has been prepared by (Sage et al, 1993).

One of the key reasons for choosing a graph based approach is the ease with which it may be modified and extended. In, for example, the parallelisation of a program targeted at a system employing a message passing model for inter-process communication, the communication primitives will have to be included explicitly in the program. This can be achieved by inserting additional nodes in the graph, known as ghost nodes.

3.2 Graph Transformation

The graph of a loop is traversed by a number of analysers which build up a picture of the loop. This picture forms part of the input to the parallelisation expert system. Basically, as loops represent a rich source of potential parallelism, the goal is to identify the best loop distribution possible. This requires the system to undertake traditional dependence analysis and reduction, followed by loop distribution. A variety of traditional and novel techniques are provided, e.g. statement reordering, loop interchange, loop skewing, variable copying and scalar and array expansion are provided as core or kernel activities.

To illustrate the principle of loop picturing consider the following trivial example:

```
          DO I = 1, N
                DO J = 1, N
s1:                   A(I,J) = B(I,J)
s2:                   C(I, J) = A(I+1, J)
                ENDDO
          ENDDO
```

Some of the facts of this loop are represented as follows:

```
(loop 1 I N)              /*  outer loop, subscript I, upper bound N */
(loop 2 J N)              /*  inner loop, subscript J, upper bound N */
(concurrent 2)            /*  only the second loop (J-loop) is parallel */
(anti 2 1)                /*  anti dependence (due to A) from s2 to s1 */
(actual A by Row)         /*  user specified                     */
(actual (B byRow)
(actual (C byRow)
(overall byRow 1)         /*  characterisation analysis has identified the*/
(overall byRow 2)         /*  overall data partition for each statement*/
```

3.3 Application of Knowledge Based Techniques

The rule-based approach used in this system receives as input a list of facts generated by the loop analysis. A set of rules have been built up as the result of studying code parallelisation. There are multiple goals, one for each transformation, and the system forward chains through the rule base until all goals are met or the system fails. However, this process can result in a number of valid transformations being selected.

The specific examples given below deal with loop interchange strategies:

```
(defrule loop-interchange-check-4
   (not (outer-loop-parallel)
   (concurrent $?front ?num $?rear)
   (test (<> ?num 1))
         =>
               (assert (apply Loop-Interchange 1 ?num))) and

(defrule loop-interchange-check-7
   (declare (salience 100)
   (not (clashing-data-distributions)
   (overall byRow 1)
   (loop 1 I ?)
   (loop 2 J ?)        =>          (assert (apply Loop-Interchange NOT REQUIRED)))
```

Both of these rules are concerned with determining whether or not a loop interchange strategy should be applied. Using the fact list derived from the trivial example in the preceding section both rules will fire giving rise to an apparent conflict. This is resolved by considering the weighting factor (salience) associated with each of the rules. For the rules described above ensuring that the partitioned iterations on each processor access local data is more important that ensuring parallel loops. This is denoted by giving the second rule a weighting of 100.

A set of rules has been developed for use with the core transformations that are implemented in the current version of the prototype. The results output from the transformation phase are passed to the generation and evaluation phase. Here a series of codes (based on the alternatives identified by the transformation selection phase) can be generated and evaluated.

3.4 Code Generation and Evaluation

The code generator accepts as input the modified graph produced by the parallelisation phase and generates lists of information representing the characteristics of this 'parallel' graph. Once again, information on the remaining data dependence, loop boundaries, variable accesses, hotspot analysis is gathered.

This information, together with some basic characteristics of the target architecture, is fed to the generator expert system (GES) which returns recommendations on initial code and data distribution.

Within the prototype the parallel architecture is regarded as a master/slave topology with an SPMD model. Inter-process communication is handled by PVM like communications strategy. Future changes will introduce the use of the MPI scheme. For demonstration purposes the current version of the prototype generates CSTools Fortran for execution on a Meiko M40.

As a program is executed profiling information is gathered. The initial distribution assumes that all slave processors will have the same execution profile. Clearly this will only be true for very simple programs and the information from the first execution will indicate which processors carry the major compute-intensive elements of the program.

Once identified, the compute-intensive element(s) can be subjected to closer scrutiny to attempt to identify the precise sections of code that are proving to be time consuming. For example problems can arise due to communication overload or external library calls. The profiler will find the subroutine(s) causing the delays and indicate the nature of the problem. This information can be used by the programmer to enable the code to be distributed in a different manner. Alternatively the information can be fed to the GES, i.e. a feedback loop is available.

The GES can handle the feedback information obtained by the profiler in several ways. Initially an attempt is made to eliminate the problem(s) in a task by recommending a different loop distribution. This will require the generator to produce an alternative partitioning of the arrays associated with the loop involved and hopefully will reduce communication times.

If this approach fails then an alternative may be to produce a different distribution of the program code across the slave processors. If this approach is adopted then the execution profiling and feedback runs again to analyse the new situation and report accordingly.

However it may be the case that having tried different loop distributions and different code partitioning strategies that no real gain in performance can be detected. If this situation arises then the GES will report this fact to the parallelisation expert system. If this step is taken then the parallelisation phase is reactivated with the goal of identifying an alternative set of transformations that will be applied to the original graph-based representation of the source program. In other words the complete development/migration cycle begins again.

4 CONCLUSION

The FortPort prototype, currently under beta-test, offers graph construction, knowledge driven loop restructuring and knowledge driven code generation. Transformations to remove or reduce data dependence are selected dynamically based on loop characteristics. Code and data is distributed across a multiprocessor architecture again on the basis of the analysis of loop and architecture characteristics.

The existing model for the creation of rules, i.e. analysis of hand coding, will be supplemented by a new approach. At the moment the transformations are expressed directly in program code. However a transformation is in effect a graph reordering function, i.e. the effect of applying a transformation to a graph-based representation of a program is simply to produce another graph. A kernel set of graph manipulations, so-called atomic operations, have been isolated. All existing transformations in the system can be expressed in terms of suitable combinations of these atomic operations. Future work will explore the effect of different combinations of the atomic operations on a program graph with the goal of identifying new transformations for parallelisation.

The major strength of the FortPort system is that it provides a complete development and migration environment. Novice users can receive expert help with the complex task of parallelising a program. Equally, experienced users can benefit from the recommendations produced by the expert systems. While the prototype will accept F77 the final version of the system will accept both F90 and HPF and will assist a user with the task of selecting appropriate parallelising statements.

5 REFERENCES

Decker, K.M., Dorvac, J.J. and Rehmann, R.M. (1993) A Knowledge-Based Scientific Parallel Programming Environment, Technical Report CSCS-TR-93-07.

Milligan, P., McConnell, R.K., Rea, S.A., Benson G. and Sage, P.P. (1992) Apparently Sequential Programming Environments for Parallel Computing, *Parallel Computing and Transputer Applications, IOS Press CIMNE*, Barcelona, 297-306.

Quill, J.C., McConnell, R.C. and Milligan, P. (1995) A Prototype Environment for Parallelization, *Lecture Notes in Computer Science*, **919**, 936-936.

Sage, P.P., Milligan, P., McConnell, R.K., Rea, S.A. and McCarney,M.T. (1993) Graph Management within the FortPort Migration Environment, *Microprocessing and Microprogramming*, **33**, 137-140.

Zima, H.P., Bast, H.J. and Gerndt, H.M. (1988) SUPERB - a tool for semi-automatic MIMD/ SIMD parallelisation, *Parallel Computing*, **6**, 1-18.

6 BIOGRAPHY

Peter Milligan is a senior lecturer in the Department of Computer Science. Currently his research programmes are devoted to the design and implementation of intelligent, semi-automated programming environments for the generation of parallel programs. In addition Dr Milligan works on the migration of mathematical codes to parallel architectures. Dr Milligan has supervised over 30 MSc and PhD students and has been involved in the organisation of six international conferences and three international workshops devoted to parallel and distributed computing. Patrick Corr is a lecturer in the Department of Computer Science. His research interests centre on the application of artificial intelligence techniques, particularly neural networks, to a range of problems in science and engineering. Paul Sage and Paul McMullan are postgraduate students currently completing PhD theses on aspects of parallel software development.

Problem-Solving on Scalable Parallel Systems Using Application Specification and Reusable Software Components

Karsten M. Decker, Jiri J. Dvorak, and René M. Rehmann
Swiss Center for Scientific Computing (CSCS/SCSC)
Via Cantonale, CH-6928 Manno, Switzerland
E-mail: {decker,dvorak,rehmann}@cscs.ch

Abstract

From the application user's point of view, ease of programming of distributed memory parallel systems has not been achieved yet. It is the purpose of this paper to demonstrate how these limitations can be overcome by our Program Development Environment PDE currently under development. The environment features a problem-oriented specification formalism and is based on a skeleton- and template-oriented application development methodology. The large set of fine-grain algorithmic skeletons and templates used in the system provides the basis for a software reuse mechanism and is managed with a knowledge-based component. Skeletons are completed with computational components by means of automatic program synthesis techniques. We present our current results, summarize the major achievements, such as software reuse and portability, and give an outlook on future research directions and related publications.

1 INTRODUCTION

Despite remarkable progress in hardware and software technology for Distributed Memory Parallel Processor (DMPP) systems over the last few years (for an overview of the state-of-the-art in parallel programming, we refer to (Decker, Dvorak, Rehmann & Rühl 1995)), simple development of scientific and engineering applications has not yet been realized.

To improve the unsatisfactory situation, it is the goal of our research to develop an easy-to-use application engineering environment (problem-solving or programming environment in the following) for the synthesis of new application software in the scientific and engineering sector in a user-centered and application driven way. Four key characteristics describe and position our approach. First, application-oriented problem description formalisms serve to focus on *what* the problem is and which computational methods should

Project funded by the Swiss National Science Foundation (SNF) in the framework of the Swiss Priority Program Informatics (SPPIF), Grant-No. SPPIF-5009-034402

be used to solve it. Second, the use of design skeletons and templates provides a software reuse mechanism and hides the difficult parts of programming DMPPs while ensuring good scalability, (efficiency-preserving) portability, and parallel efficiency. Third, interactive guidance supports exploitation of user knowledge as completely as possible. Finally, automatic program synthesis techniques ensure a transparent coding process.

Based on the recent success of parallel systems in the business sector, where familiar application development interfaces are applied and parallelism is offered transparently, we believe that a descriptive formalism is essential. Algorithmic skeletons or similar structures were proposed in the past for software engineering and reuse (Waters 1982) as well as for parallel computations (Cole 1989). The basic idea underlying these approaches is to encode reusable structural characteristics of algorithms in skeletons. A skeleton typically contains open, i.e., *generic*, parts that have to be filled in to adapt the skeleton to the given situation and to get a complete algorithm or algorithm component.

It is the purpose of this paper to outline the achievements of our high-level *Program Development Environment* (PDE) and to report on future research directions. For a detailed description of the design objectives, the underlying application development methodology, and an assessment of the methods we use, the interested reader is referred to (Decker, Dvorak & Rehmann 1994*b*).

2 RESULTS

2.1 A Programming Environment for Stencil-based Problems

In the spirit of a rapid prototyping approach to test system functionality, we started our research with feasibility studies for the very simple class of stencil-based problems (Decker, Dvorak & Rehmann 1994*a*) characterized by the operation of a local computational stencil on n-dimensional grids. Applications in this class include, for instance, the restoration of gray-scale images with different smoothing operators and the solution of partial differential equations according to the finite difference method.

We illustrate the functionality of the PDE for this problem class with the solution of the Poisson equation on a simple two-dimensional rectangular grid with periodic boundary conditions. The Laplace operator is approximated by the nearest-neighbor, symmetric 5-point stencil and we use a Gauss-Seidel algorithm with two colors to accomplish the iterative solution.

The declarative problem description can be done graphically with the *Stencil Modeling Programming Assistant Interface* (SMPAI, Fig. 1) which is then translated into the textual *Stencil Problem Specification Language* (SPSL) description shown in Fig. 2. If this is preferred by the user, the programming task can start directly in SPSL.

As can be seen, the problem description consists of the specification of the problem type, the geometry of the problem domain, the size and dimensionality of the grid, the structure of a grid cell, the boundary conditions for each physical boundary of the grid, the computational stencil, the numerical method, and the domain decomposition scheme. Since SPSL realizes a complete problem description (Roth 1993), there is no further user interaction with the PDE required.

The PDE now successively transforms the SPSL problem description into a compil-

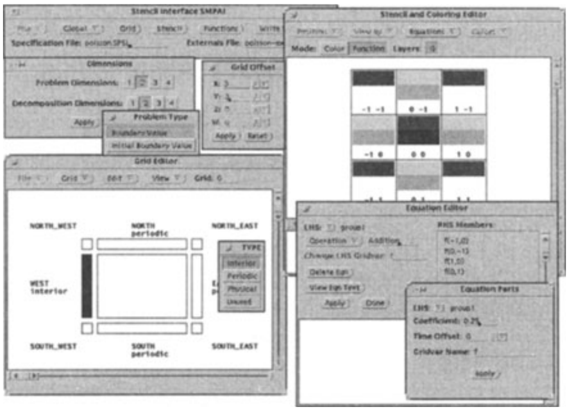


Figure 1 The graphical user interface of the SMPAI.

```
grid_spec {
  grid_offset = [0,0];
  grid_size = {512,512};
  boundary_grid {
    boundary_grid_type = PERIODIC_BND;
  } SOUTH;
  boundary_grid {
    boundary_grid_type = PERIODIC_BND;
  } NORTH;
  .......
};
stencil_spec {
  stencil_action = {
    f<[0,0]> = 0.25*(f<[-1,0]> + f<[0,-1]> + f<[1,0]> + f<[0,1]>)
                 - f_rho<[0,0]>
               };
};
```

Figure 2 SPSL description of the programming example.

able program. The *Programming Assistant* (PA) first reads the problem representation and starts the reasoning process with the goal to find the most appropriate algorithmic skeleton for the given problem. The reasoning process is a rule-driven descent in the skeleton hierarchy, based on the information given in the problem specification and various rule bases maintained by the PA. Expertise about the application domain, parallel programming, and software engineering are encoded in the rules that control this skeleton selection. Knowledge about characteristics of different hardware platforms will be added in a future prototype. Together with the specification of the problem at an abstract level, the hardware knowledge integrated in the skeleton selection will ensure portability at the algorithmic level, above the level of general-purpose programming languages.

The chosen skeleton contains all information needed for the generation of a parallel framework suitable for the problem under consideration. In particular, it defines the data distribution scheme and the communication and synchronization structure. Based on the chosen skeleton and the problem representation, the PA produces two different outputs for the two-component structure of the *Program Synthesizer* (PS).

One part of the final code generation step is done by the TINA skeleton generator (Gutzwiller 1993), which generates the C-code for setting up the process topology, data distribution and communication calls for different message-passing interfaces, and calls the computational components.

The computational components are generated by the other component of the program synthesizer PS (Rehmann 1994). This component starts from an abstract definition of the computational units of the application and the definition of the grid structure. Using this input, it generates code for the function which calculates the user-defined computational stencil, the functions for filling and scattering the communication buffers from and to the grid, and the function calls for the various types of boundary conditions.

2.2 Towards Programming of General Data-parallel Problems

From the application user's point of view, the problem domain of stencil-based applications discussed in Sect. 2.1 is of rather low importance. To qualitatively increase the usefulness and attractivity of the PDE, our most recent and current research is concerned with a major step towards supporting the programming of general data-parallel problems which are of real practical interest to the scientific user community.

To achieve this goal, we follow a step-wise approach. Analyzing user requirements and identifying the important components of real applications, we first focus on problems resulting in the formulation of linear algebra operations. Within this problem class, we concentrated on developing a programming environment for iterative solvers for general linear systems.

The problem class of iterative solvers for general linear systems has very different characteristics than the class of stencil-based problems: the problem description may be incomplete, problem realization may require more than one algorithmic skeleton or template with a fixed parallel structure, and consequently, the careful design of data structures becomes crucial.

To realize a programming environment for this problem class, all three functional components of the PDE, i.e., the PAI, the PA, and the PS, need to be reconsidered and enhanced. In this paper, we report on three activities: the development of the specification language for the problem class of iterative solvers, a first prototype for the corresponding *Data Modeling Programming Assistant Interface* (DMPAI), and a related prototype of the PA.

An essential part of the DMPAI is the underlying declarative problem specification language. The *Basic Language for Iterative, Parallel Solvers* (BLIPS) (Toupin 1994) has a Pascal-like syntax, provides support for intrinsic and user libraries, supports abstract property specification to describe problem characteristics, and has dynamic language support. The latter characteristic of BLIPS allows the user to define and adapt the language

```
template solve(A : matrix; x, y : colvector);
  A is symmetric and pos_definite;
  for solve A * x = b;
    do
      { implementation of the solver algorithm (e.g., CG)}
    end do;

template solve(A : matrix; x, y : colvector);
  A is non_symmetric and transp_not_avail and storage_limited;
  for solve A * x = b;
    do
      { implementation of the solver algorithm (e.g., BiCGSTAB)}
    end do;

procedure main;
  var   A: matrix;
      x,b: colvector;
  where A is symmetric and pos_definite;
  do
    read A from "pde.mat";
    read b from "pde-b.vec";
    solve A*x=b;
    write x to "pde_soln.vec";
  end do;
```

Figure 3 Definition of a linear solver in BLIPS.

according to his specific needs. An example showing how to define a linear solver for a specific type of matrix is given in Fig. 3.

The most recent prototype contains the static knowledge base for the PA, i.e., templates implementing different algorithms for parallel iterative solvers. These templates are wrappers for a library of parallel iterative solvers developed in another project at CSCS. Additionally, we have extended the dynamic knowledge, i.e., the rules to make use of the correct templates for a specified application and we have developed a user interface that allows the specification of an application and the editing of new or existing templates. As the templates only contain calls to library functions, no sophisticated program synthesizer is needed.

3 FUTURE DIRECTIONS

Our future research will concentrate on improved programming environments for the class of data-parallel programs. We intend to successively relax the constraints on the supported application spectrum currently imposed. Formalization of the large amount of application knowledge to enhance the rule base of the PA will be crucial to ensure the long-term success of these systems.

An important topic which will be investigated is which requirements the user dialog with the PDE must satisfy to guarantee successful interaction, respecting (and taking advantage of) the conceptual models, knowledge structures, and working processes of our target user community, i.e., application users.

Another subject of research is the development of a hardware knowledge base with corresponding hardware-dependent algorithmic skeletons and templates. Together with appropriate rules to choose the correct skeleton or template for a given hardware, it will be possible to generate program code which runs optimally on specific hardware. This mechanism ensures true, i.e., efficiency-preserving portability across hardware platforms as the application-oriented problem description needs not be changed to optimally run an application on different hardware platforms.

A further important topic which we believe should be addressed in the near future is teaching effective usage of DMPPs. Here we envisage machine-assisted learning which could be realized more or less easily by suitable enhancements of our PDE.

Finally, the supported application spectrum should be broadened further, in particular with support for non-numerical problems.

REFERENCES

Cole, M. (1989), *Algorithmic Skeletons: Structured Management of Parallel Computation*, Research Monographs in Parallel and Distributed Computation, The MIT Press, Cambridge, MA, USA.

Decker, K. M., Dvorak, J. J. & Rehmann, R. M. (1994*a*), A Knowledge-based Scientific Parallel Programming Environment, *in* K. M. Decker & R. M. Rehmann, eds, 'Working Conference on Programming Environments for Massively Parallel Distributed Systems', Birkhäuser Verlag, Basel, pp. 127–138.

Decker, K. M., Dvorak, J. J. & Rehmann, R. M. (1994*b*), User-driven development of a novel programming environment for distributed memory parallel processor systems, *in* 'Priority Program Informatics Research Information Conference Module 3 Massively Parallel Systems', Swiss National Science Foundation, pp. 40–47.

Decker, K. M., Dvorak, J. J., Rehmann, R. M. & Rühl, R. (1995), 'Matching User Requirements in Parallel Programming', *Future Generations Computer Systems*. accepted for publication.

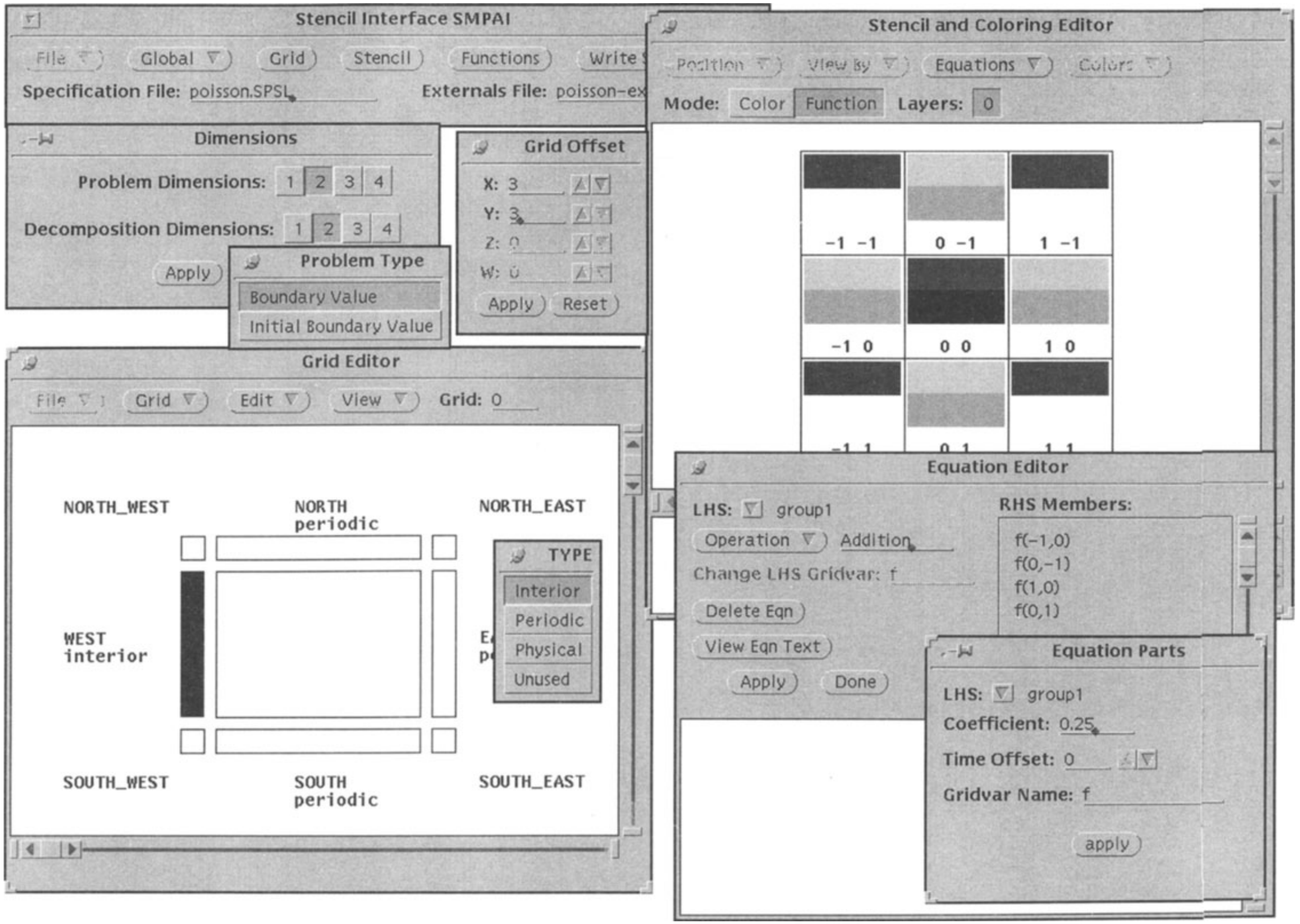
Gutzwiller, S. (1993), Werkzeuge und Methoden des skelettorientierten Programmierens von Parallelrechnern, PhD thesis, University of Basel. In German.

Rehmann, R. (1994), Automatic Generation of Programs for a Scientific Parallel Programming Environment, Technical Report CSCS-TR-94-02, Centro Svizzero di Calcolo Scientifico, CH-6928 Manno, Switzerland.

Roth, M. (1993), Generation of Algorithmic Skeletons from Stencil Specifications, Master's thesis, IAM, University of Bern. In German.

Toupin, T. (1994), **B$^{I}_{L}$PS** Language Specification Proposal, Technical Note SeRD-CSCS-TN-94-09, Swiss Scientific Computing Center, CH-6928 Manno, Switzerland.

Waters, R. C. (1982), 'The Programmer's Apprentice: Knowledge Based Program Editing', *IEEE Trans. on Software Eng.* **SE-8**(1), 1–12.



Specification File: poisson.SPSL

Externals File: poisson-ex

Problem Dimensions: 1 [✓ 2] 3 4

Decomposition Dimensions: 1 [✓ 2] 3 4

Problem Type: [✓ Boundary Value] Initial Boundary Value

X: 3
Y: 3
Z: 0
W: 0

Grid: 0

NORTH periodic

WEST interior

SOUTH periodic

Mode: Color [✓ Function] Layers: 0

LHS: group1

Operation: Addition

Change LHS Gridvar: f

RHS Members:
f(−1,0)
f(0,−1)
f(1,0)
f(0,1)

LHS: group1

Coefficient: 0.25

Time Offset: 0

Gridvar Name: f

30
The PS project: development of a simulator of PVM applications for Heterogeneous and Network Computing

R. Aversa[a], A. Mazzeo[a], N. Mazzocca[a] and U. Villano[b]

[a]DIS, Universita' di Napoli, Via Claudio 21, 80125 Napoli (Italy)
[b]IRSIP-CNR, Via Claudio 21, 80125 Napoli (Italy)
e-mail: [aversa,mazzeo,mazzocca,villano]@nadis.dis.unina.it

Abstract

Heterogeneous computing environments require performance evaluation techniques that are more sophisticated and cost-effective than those currently used. This paper briefly describes a project aimed at the development of PS, a simulation environment for the performance analysis of distributed applications executed in Heterogeneous and Network Computing environments through the PVM run-time system.

Keywords

Heterogeneous Computing, Network Computing, Simulation, Performance Evaluation, PVM.

1 INTRODUCTION

Heterogeneous Computing (HC) (Khokhar, 1993), (Mechoso, 1994) and Network Computing (NC) (Anderson, 1995) share as a common denominator the exploitation of heterogeneous computing resources. The increased complexity of heterogeneous environments calls for performance measurement and analysis techniques that are more sophisticated and cost-effective than those commonly used in homogeneous parallel and distributed computing systems. Of great practical interest in particular is to obtain performance data before the software implementation, since this enables the software developer to choose carefully the workload to be assigned to each target machine as early as in the software design stage. This is a particularly thorny problem both in HC (Wang, 1992) and in NC (Mazzeo, 1995).

The above considerations are among the premises of the PS project, started in 1993 as a cooperation between the Department of Informatics of the University of Naples and IRSIP, an institute of the Italian National Research Council (CNR). PS (*PVM Simulator*) is a simulator for the performance analysis of distributed applications based on the Parallel Virtual Machine paradigm, the *de facto* standard for programming HC and NC systems (Geist, 1994). The software simulated by PS can either be a complete PVM program or a *prototype*, i.e., a partially implemented program design. The simulation of the whole hardware/software system makes it possible to obtain aggregate and analytical indexes related to the heterogeneous system performance (e.g., efficiency, throughput, response time, individual processor utilisation), or traces which can be processed off-line by *Paragraph* to visualise the simulated program execution in a variety of different views.

In our opinion, the role that simulation techniques can play in parallel software engineering has not yet been fully recognised. In a recent paper, we have shown that simulation tools can help managing the complexity of software development for heterogeneous hardware (Aversa, 1995a). In this context, the fundamental advantage of simulation is flexibility. Simulation tools make it possible to compare the behaviour of different algorithms on the same hardware platform, to assess the effect of different problem decompositions, task allocations and load sharing techniques, or even to study the performance of a single algorithm on several existing or hypothetical computing environments. Simulation tools require neither the oversimplifications which are commonly used to deal with complex hardware/software systems through analytical models, nor the availability of fully-developed software and of a real machine, which are necessary for the performance analysis by monitoring/tracing tools. On the minus side, it should be noted that accurate simulations are computationally expensive. PS is characterized by a light simulation overhead, thanks to the adoption of models of program tasks which are at a higher level of abstraction than those adopted by other existing simulators. The modelling of the communication subsystem is instead particularly accurate, and is also capable of considering the effect of external network load. The fairly high accuracy attained in all validation tests of the simulator show that these are perfectly reasonable solutions, at least for the heterogeneous computing platforms which are currently available.

2 PS: A PVM SIMULATOR

PS (*PVM Simulator*) makes it possible to simulate a complete PVM application by combining predefined objects that simulate the inter-task communication subsystem (PVM daemons, TCP/IP protocol, network interfaces, physical interconnection medium) along with objects simulating the tasks making up the user program (Aversa, 1994-1995b). Its simulation kernel has been implemented using the *Ptolemy* environment (Buck, 1994), and can therefore be ported to any computing platform where Ptolemy can be executed, e.g., Sun, DEC and HP workstations, and IBM PC under Linux. A complete Ptolemy application (called a *Universe*) consists of a network of interconnected Blocks. Blocks may be either *Stars* (atomic objects) or *Galaxies* (composite objects, made up of Stars and other Galaxies). PS provides predefined objects (Stars and Galaxies) that model the computing environment. These objects are to be connected using the Ptolemy interactive graphical interface to Galaxies modelling user code, thus setting up a Universe which is representative of a complete PVM application. The effect of network load can be taken into account by means of predefined Stars sharing the same physical network medium, which generate a given statistical network load distribution.

A typical PS simulation session consists of the following phases (Figure 1):

1) The Galaxies modelling user code are built. User code can be supplied either as a complete PVM program or as a *prototype*. Prototypes are skeletons of code containing PVM calls, where some of the computations have been replaced by calls to delay procedures taking into account the time spent in the actual code.

2) The user sets up a Universe that is representative of the whole hardware/software system by constructing a graph where the nodes are icons corresponding to Galaxies or Stars, and the (directed) arcs represent interactions between them.
3) A suitable number of monitoring Stars, which record the events occurring in a specific point of the Universe, are inserted into the Universe diagram.
4) The modules of simulation code corresponding to the icons contained in the Universe are compiled and linked to the event scheduler.
5) The simulation is performed by executing the program produced at point 4.
6) The information on system performance produced during the simulation is examined. As Stars which produce a trace of program execution in a format compatible with *Paragraph* (Heath, 1991) are provided, it is also possible to collect more in-depth information about program behaviour and system performance off-line. If the measured performance is not satisfactory, new hardware configurations or allocation strategies have to be tried. This entails repeating all the previous steps several times.

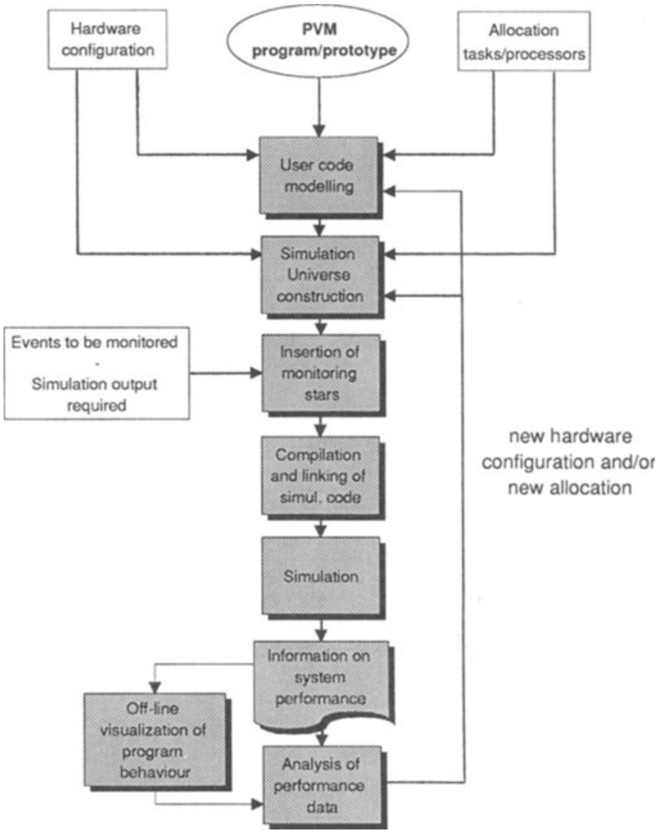

Hardware configuration
PVM program/prototype
Allocation tasks/processors
User code modelling
Simulation Universe construction
Events to be monitored - Simulation output required
Insertion of monitoring stars
Compilation and linking of simul. code
Simulation
Information on system performance
Off-line visualization of program behaviour
Analysis of performance data
new hardware configuration and/or new allocation

Figure 1 Phases of a PS simulation session.

A point that is worth discussing in more detail is how the program to be simulated is modelled in PS. The PVM application behaviour is modelled at user-task level by a set of *UT* (User Task) Galaxies, which model the user tasks running on every processor. For the sake of simulation accuracy, these objects should reproduce as closely as possible the sequence and the timing of the run-time requests of the actual program to the PVM daemon. The PS environment provides two different methods for the construction of *UT* Galaxies. The first possibility is to adopt a *delay-PVM call* model, building a graph where Stars invoking PVM primitives are interleaved by delay Stars corresponding to the time spent executing the code contained between them (the computation delays are estimates found by direct measurement under suitable test conditions or through statistical and/or analytical models). Another possible approach relies on the examination of the source code by a static program analysis tool, which analyses the performance behaviour of each task on the particular sequential or parallel computer where it will be actually executed. A tool for the static analysis of PVM code within the PS framework is currently under testing, and will be released in the near future.

The second option is to use *execution-driven* simulation (Convington, 1988), which is surely the recommended option when program execution time has not simple dependence on input data. In execution-driven mode, every PVM program task is modelled by a *UT* Star containing a version of the user code where PVM calls have been replaced with calls for service to the objects simulating the Parallel Virtual Machine level. During the simulation, the PVM program is *actually executed* (i.e., it is not simulated) in quasi-concurrence on the workstation hosting the PS environment. Whereas, the behaviour of the PVM run-time support, of the communication protocols and of the interconnection network is still simulated. Unlike other simulators based on the execution-driven approach (Davis, 1991), (Brewer, 1991), PS uses a simplified method to estimate the execution time of the block of code between two successive interactions with the simulation engine (interactions occur at every PVM call). This method introduces a much lower simulation overhead and leads to no significant accuracy loss, due to the coarse granularity of PVM tasks.

Figure 2 shows a screen dump taken during one of the simulation sessions described in (Aversa, 1995a). The picture shows the host display during the simulation of one of the algorithms considered for performing a matrix multiplication on a network of four workstations. The large window in the middle of the picture is the graphical representation of the simulation Universe. The four icons on the left of the Universe are the *UT* Galaxies simulating the behaviour of the application code executed on every node of the network. The Parallel Virtual Machine level, which models a computing node along with its run-time support, consists of the column of four icons in the middle of the window. The Network level is made up of the single icon on the right, which represents the shared transmission medium. The window in the lower left corner is an insight view of a *UT* Galaxy, showing in some detail its interface to the Galaxy simulating the PVM daemon. Figure 2 also shows samples of the Paragraph performance summaries of the simulated program execution, namely a Spacetime diagram and a Utilization Gantt chart.

3 CURRENT STATUS OF THE PROJECT - FUTURE DIRECTIONS

Over the last year and half PS has evolved from a largely-incomplete prototype to a fully working version. The results of all our validation tests, some of which are reported in (Aversa,

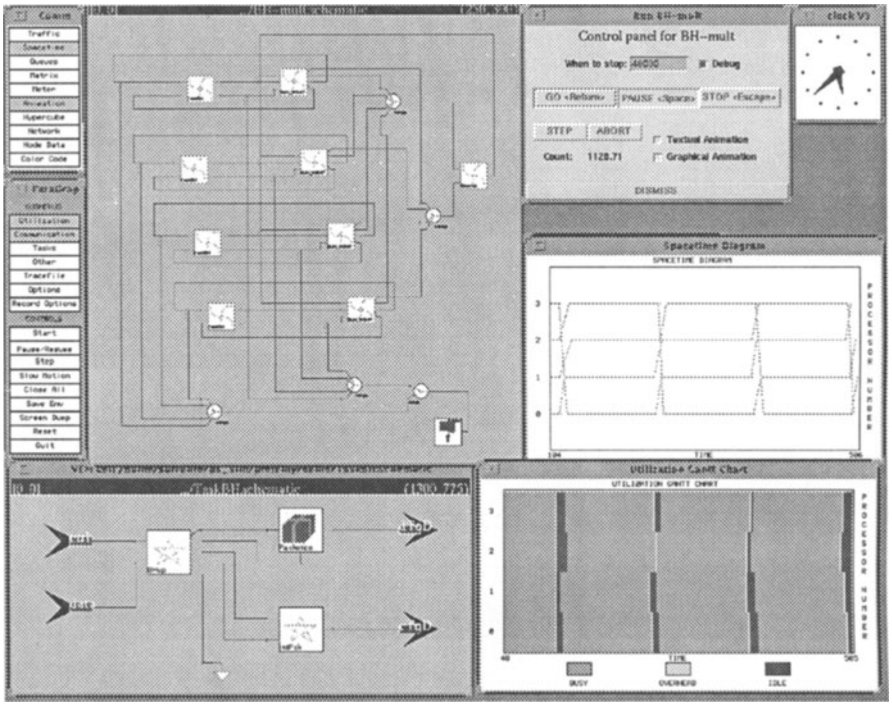


Figure 2 Host display during a PS simulation session.

1994), (Aversa, 1995b), have been particularly encouraging. PS never led to figures more than 5% far from those obtained by running the actual program on the real computing environment. Using the delay-PVM call model for user task, the PS simulation speed is too high to be compared to the execution rate of the actual program on a single processor. Whereas, if execution-driven mode is chosen, the execution of a PVM program is emulated a factor of only 6 to 9 times slower than the execution of the same program in quasi-concurrence on the simulation host. Unfortunately, we are not equally satisfied of the ease of use and friendliness of PS. The construction of the delay-PVM call Galaxies modelling user code is not easy and requires a sufficient degree of familiarity with the simulator itself. Furthermore, setting up the simulation Universe using the Ptolemy graphical interface is a time-consuming task, and one that does not lend itself to be automated by any software tool. This is the reason why we decided not to release the first version of PS outside our research group, and to redesign the simulator user interface in order to produce a new version amenable to be publicly distributed.

At the state of the art, the core of the version 2.0 of PS has been already implemented and is currently under testing. Besides the execution-driven mode (which is still supported, since it seems the best option for expert users), the new version of PS also makes it possible to launch a simulation session without using the Ptolemy graphical interface. In this case, the simulator is fed with a trace previously obtained by executing the PVM program in quasi-concurrence on a single workstation or in real concurrence on a scaled-down distributed environment, and tracing the calls to the run-time support by means of a PVM-tracing library. A further issue that is being addressed in the development of the second version of PS is the possibility to study the effect of load due to tasks not belonging to the program to be analyzed. This will be obtained in much the same way as the effect of external network load is taken into account, i.e., by the addition of statistical interference with the computations on-going in each node.

4 REFERENCES

Anderson, T.E., Culler, D.E. and Patterson, D.A. (Feb. 1995) A Case for NOW (Networks of Workstations). *IEEE Micro*, **15**, 54-64.

Aversa, R., Mazzocca, N. and Villano, U. (1994) PS: a Simulator for Heterogeneous Computing Environments, in *Massively Parallel Processing Applications and Development* (eds. L. Dekker, W. Smit and J. C. Zuidervaart), Elsevier, 335-343.

Aversa, R., Mazzeo, A., Mazzocca, N. and Villano, U. (1995) The Use of Simulation for Software Development in Heterogeneous Computing Environments. *Proc. Int. Conf. on Par. and Distr. Processing Techniques and Applications*, Athens, GA, 581-590.

Aversa, R., Mazzocca, N. and Villano, U. (1995) Design of a Simulator of Heterogeneous Computing Environments. to be published in *Simulation Practice and Theory*.

Brewer, E.A., Dellarocas, C.N., Colbrook, A. and Weihl, W.E. (1991) PROTEUS: a High-performance Parallel-architecture Simulator. *Tech. Rep. MIT/LCS/TR-516*, Cambridge, MA.

Buck, J.T., Ha, S., Lee, E.A. and Messerschmitt, D.G., (1994) Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *Int. Journal of Computer Simulation*, **4**, 155-182.

Convington, R.C., Madala, S., Mehta, V., Jump, J.R. and Sinclair, J.B. (1988) The Rice Parallel Processing Testbed. *Proc. 1988 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 4-11.

Davis, H., Goldschmidt, S.R. and Hennessy, J. (1991) Multiprocessor Simulation and Tracing using Tango. *Proc. 1991 Int. Conf. on Parallel Processing*, II99-II107.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. (1994) *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, MA.

Heath, M.T. and Etheridge, J.A. (Sept. 1991) Visualizing the Performance of Parallel Programs. *IEEE Software*, **8**, 29-39.

Khokhar, A.A., Prasanna, V.K., Shaaban, M.E. and Wang, C. (June 1993) Heterogeneous Computing: Challenges and Opportunities. *IEEE Computer*, **26**, 18-27.

Mazzeo, A., Mazzocca, N. and Villano, U. (1995) Efficiency Measurements in Heterogeneous Distributed Computing Systems: from Theory to Practice. submitted to *Concurrency: Practice and Experience*.

Mechoso, C.R., Farrara, J.D. and Spahr, J.A., (Summer 1994) Achieving Superlinear Speedup on a Heterogeneous, Distributed System. *IEEE Par. and Distr. Technology*, **2**, 57-61.

Wang, M., Kim, S., Nichols, M.A., Freund, R.F., Siegel, R.F. and Nation, W.G. (1992) Augmenting the Optimal Selection Theory for Superconcurrency. *Proc. Workshop on Heterogeneous Processing*, IEEE Computer Society Press, 13-21.

31

Supporting integrated modelling of parallel hybrid systems

C. I. Birkinshaw and P. R. Croll
Department of Computer Science
University of Sheffield, UK
email: carl@dcs.shef.ac.uk

Abstract
PERCH is the name of a software tool specifically designed to capture the requirements of hybrid systems. The aim is to provide an integrated environment from requirement capture through to implementation of real applications that are complex and hybrid in nature. The tool encourages a parallel view of the systems being modelled and acts as a high level interface to further parallel design and analysis environments.

Keywords
Hybrid systems, Petri nets, temporal logic, parallel software engineering

1 INTRODUCTION

The Mixed-Mode* project, a collaboration between two departments at the University of Sheffield, is concerned with realistic modelling of complex hybrid systems (Fahrland, 1970) comprising of both continuous and discrete components. When the relative mix of continuous dynamics and discrete events is non trivial, then the hybrid systems used to model them can be difficult to handle. Conventional modelling separates the discrete and continuous aspects, which simplifies the problem of modelling and analysis. The disadvantage to this approach is that the lack of realistic modelling introduces inefficiencies due to lack of interaction between the separated components. In many real practical systems, it is impossible to separate the discrete and continuous parts because they are strongly connected in either the space domain or the time domain. Handling a system in an integrated fashion would give us a better understanding of the system's behaviour.

The real world is intrinsically parallel and the hybrid systems being modelled should reflect that parallelism as realistically as possible. To achieve this the methods and tools used in our project both incorporate explicit parallel constructs and permit analysis of parallel behaviour.

* EPSRC funded grant number GR/H/73584 "Strategic Planning and Operational Control for Complex Mixed-mode Systems", Prof DA Linkens, Prof WML Holcombe, Prof S Banks, Prof PJ Fleming, Dr N Mort, Dr PR Croll.

The purpose of the tool described here is to give a means of capturing the essential requirements of complex hybrid systems, providing a respository of information that can be utilised in either of the two interpretations used in the Mixed-Mode project so far. Presently, the features of hybrid systems are represented using Extended Coloured Petri Nets (ECPN) (Yang et al., 1994) or Hybrid Projection Temporal Logic (HPTL) (Duan, 1994). The result is a tool called PERCH (Prototype Environment for Requirement Capture of Hybrid systems). PERCH itself adopts a slightly higher level (and sufficiently less formal) model which aims to capture the requirements of a hybrid system, giving a specification that is sufficient to allow the generation of Petri net or temporal logic based models of the same system. Figure 1 shows an overview of the PERCH environment. PERCH consists of a graphical interface tied to a relational database. Details of the requirements and specification of a hybrid system are entered into the database, and elements of the data are extracted to produce specifications in either HPTL or ECPN.

Petri nets are a mature model of concurrency, and high level nets such as Coloured Petri nets (Jensen, 1990) are powerful modelling tools which now have commercial software support (MetaSoft, 1992). Petri nets are especially useful, as they allow us to model the whole system; not just the software or hardware which is under computer control, but also the non-deterministic environment in which the system operates. The Hybrid Projection Temporal logic allows the modelling of continuous and discrete time and introduces a parallel operator. As such it can express the temporal characteristics of parallel hybrid systems.

PERCH maintains integrity constraints on the contents of the database and provides export functions which generate output suitable for the HPTL or ECPN environment.

Section 2 gives an overview of the PERCH model, while Section 3 outlines the different interfaces presented to the system developer. In Section 4 the Design/CPN and HPTL tools are presented.

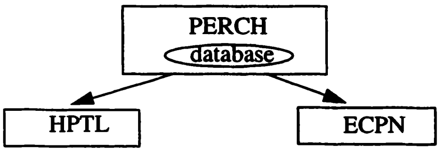


Figure 1 PERCH Overview.

2 THE PERCH SPECIFICATION MODEL

In PERCH, A hybrid system is described in terms of objects that may reflect either a logical view of the system, or actual physical entities. Each object consists of a number of states which describe the discrete events of an object, and continuous functions which describe the continuous behaviour of the system for each state. This model loosely follows the conventions of the Hybrid Machine model (Duan et al., 1995). In addressing the issue of complexity, a hierarchical approach is adopted here. This borrows from the concept of the Hierarchical Hybrid Machine of Duan (Duan, 1995), though its semantic treatment is less formal.

A PERCH object is a physical or logical entity in a hybrid system. Objects may themselves contain further objects, providing a hierarchical view of a system. An object that contains references to objects at a lower level is called a *machine* in PERCH terminology. Each object is described in terms of state changes and continuous functions which operate between states.

2.1 States and functions

A state holds over the lifetime of a continuous function, where variables are being continually updated according to that function. Entry to the state occurs when a precondition is satisfied. Exit from the state occurs when a leaving condition or postcondition is satisfied. When a postcondition is satisfied the object moves to another named state. At this point, discrete variables may be assigned new values. An assignment may include distributed assignments, using the input (?) and output (!) operators of CSP (Hoare, 1978) and named channels which are declared in the machine-wide attributes section. All assignment is assumed to occur in parallel, so that the ordering of input and output statements will not potentially cause deadlock. These communication channels are implemented in the same manner as that described in (Birkinshaw and Croll, 1995) which also describes how livelocks and deadlocks can be avoided in synchronous message passing systems. The same postcondition may name more than one next state, nominating a set of states to be executed in parallel.

2.2 Variables and abbreviations

Variables are either discrete or continuous, they are given a datatype and may be local to an object or shared between objects and machines.

Abbreviations are used as a mechanism for refinement. For example, the predicate *too-hot* may be used as a precondition to a state that is later defined in a sub-object as *temp>500 and fan=off*. Therefore the initial sketch of the system can contain predicates expressing a plain English goal or constraint such as *abnormal_conditions* which is later refined through substitution to a mathematically precise definition. Objects naturally inherit the abbreviations of their lower level objects.

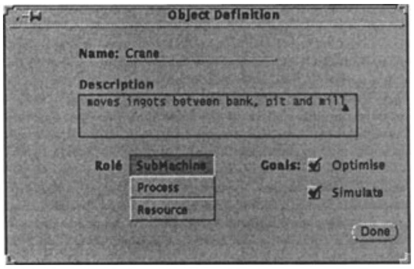
2.3 Machine-wide attributes

The properties described so far have been object specific, i.e. they relate to the local object only. There are also a number of properties that are described as being machine-wide, as they are inherited automatically by the objects of a machine. These include: a list of variables shared by sub-objects; abbreviations used in object definitions and constraints which are HPTL expressions expected to hold true throughout the lifetime of a machine.

3 TOOL INTERFACE

The PERCH interface is X-windows OpenLook compliant, and the screenshots which follow demonstrate some its features. From Figure 2 one can see that an object has a name and a free-form text description. The user can select the object to be a machine - i.e. this object is a container for a number of other objects, or a process or resource. A process captures the idea of a logical object. A resource refers to a physical device or entity in the system. The difference in meaning is currently only conceptual.

The state editor is shown in Figure 3. Conditions may include temporal logic operators, continuous functions have access to differential and integral functions. A list of states is given for the currently selected object. The observable periodicity is an indication of how often a discrete event should observe the latest values of continuous variables.



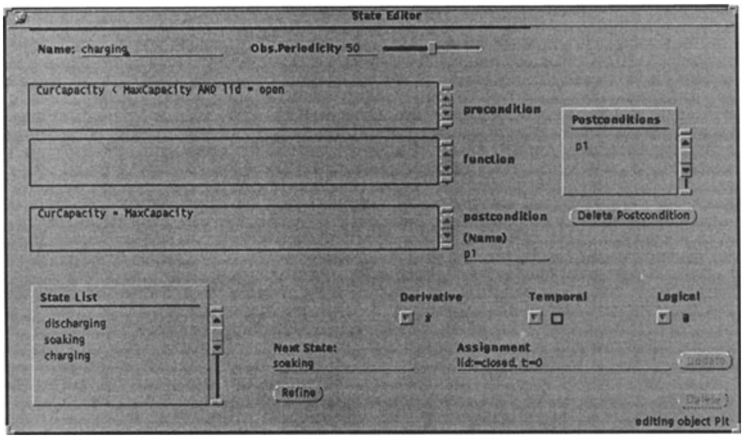
Name: Crane

Description: moves ingots between bank, pit and mill

Role: [✓ SubMachine] Process Resource

Goals: ✓ Optimise ✓ Simulate

Figure 2 Object editor.



Name: charging

Obs.Periodicity 50

precondition: CurCapacity < MaxCapacity AND lid = open

function:

postcondition: CurCapacity = MaxCapacity

Postconditions: p1

(Name) p1

State List: discharging, soaking, charging

Derivative | Temporal | Logical

Next State: soaking

Assignment: lid:=closed, t:=0

editing object Pit

Figure 3 State editor.

4 AUTOMATIC MODEL GENERATION

PERCH provides export functions which produce files suitable for importing into the target environments of either HPTL or Design/CPN. These files contain instructions for automatically generating a model from the PERCH specification.

Design/CPN (MetaSoft, 1992) is a commercial CASE environment that supports coloured Petri nets, and is the environment used for building ECPN models. Each PERCH object is constructed as a Design/CPN sub-net. A number of states are collected together in each sub-net. Discrete variables are represented by a named place containing the current value of the variable. Global or shared variables are represented by fusion places which may span several nets. As a state is generated, some glue is needed to connect a state with its previous and next state, and this requires a few extra places or transitions to be inserted between the main transitions that describe the activity occurring within each state.

Continuous variables are modelled in CPN by a place which holds all the continuous variables local to that state. The place contains a multiset of values represented as tuples of the form (variable_tag, variable_value). This is similar to the notion of an environment in E/R nets (Ghezzi et al., 1991). The tag is used to bind the correct tuple values to variables in arc inscriptions. PERCH creates ML output files in the current directory. These files may be imported into the Design/CPN tool, and rely on a pre-written library of support code designed especially for generating Petri net pages from PERCH specifications. Colour sets are automatically enumerated, variables typed and abbreviations declared as value constants.

Although no machine support yet exists for executing Hybrid Projection Temporal Logic, PERCH can still support the annotation of HPTL predicates and constraints in the semi-formal tool specification, and then perform some simple parsing and integrity checks on these annotations (i.e. check that variables in predicates are within the scope of the current object). To this end, the tool allows underspecification, where a state may be left with its behaviour only partially defined, to be refined at a later date into a complete state or sub-object.

4.1 Demonstration Example

Space precludes description of any large examples in this paper, a model of a steel production process has been given in (Yang et al., 1995). This, together with a simpler example of a thermostat controller are configured for demonstration purposes on a Sun portable. The generated net of the thermostate controller is shown in Figure 4.

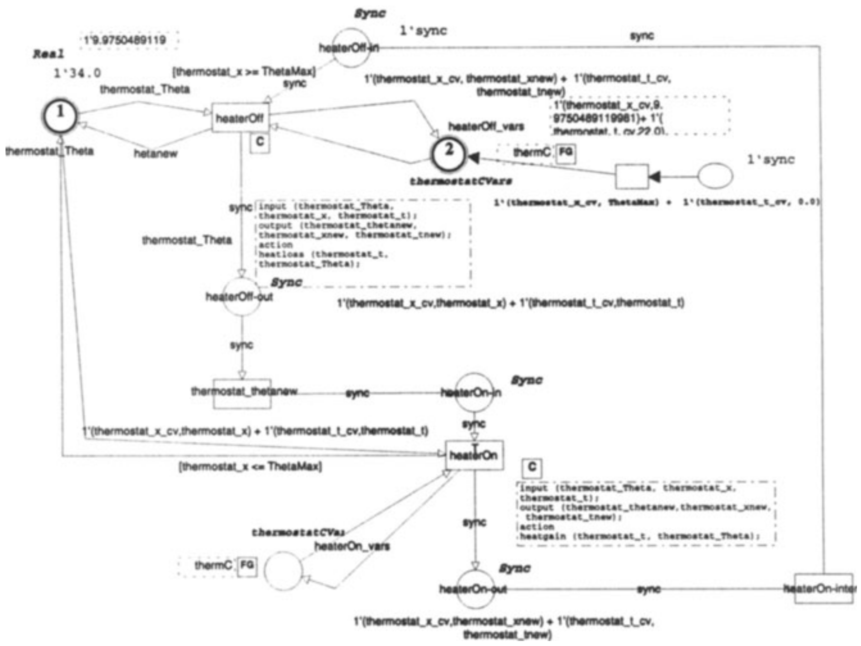

Sync
1'sync
heaterOff-in
sync
1'9.9750489119
Real
1'34.0
[thermostat_x >= ThetaMax]
thermostat_Theta
heaterOff
C
thermostat_Theta
hetanew
heaterOff_vars
thermC FG
thermostatCVars
1'sync
input (thermostat_Theta, thermostat_x, thermostat_t); output (thermostat_thetanew, thermostat_xnew, thermostat_tnew); action heatloss (thermostat_t, thermostat_Theta);
heaterOff-out
thermostat_thetanew
heaterOn-in
heaterOn
[thermostat_x <= ThetaMax]
input (thermostat_Theta, thermostat_x, thermostat_t); output (thermostat_thetanew, thermostat_xnew, thermostat_tnew); action heatgain (thermostat_t, thermostat_Theta);
heaterOn_vars
heaterOn-out
heaterOn-inter
1'(thermostat_x_cv,thermostat_x) + 1'(thermostat_t_cv,thermostat_t)
1'(thermostat_x_cv,thermostat_xnew) + 1'(thermostat_t_cv, thermostat_tnew)

Figure 4 Petri net of thermostat controller.

5 CONCLUSIONS

Formal methods and languages, such as Petri nets and temporal logics can be extended to describe the continuous and discrete behaviour of hybrid systems. Such systems are inherently parallel, in that the continuous steps and discrete events are viewed as happening concurrently and the physical and logical processes being modelled are typically distributed. However, it is difficult to capture all the necessary requirements and conditions of hybrid systems using a single specification language, given the complexity and evolving nature of requirements. A higher level, less formal method of capturing the salient points of hybrid systems has been described here, one which with machine support, is capable of massaging the requirements into one of several formal specification formats. In the case of extended Petri nets, computer generation of an executable model is possible.

Because the software environment itself is complex, computer support was needed for managing the different formalisms used in this project. PERCH has been designed to meet that requirement and give computer scientists and control engineers a common platform for specification and discussion. The tool is extensible in as far as new fields and database tables can be added without interfering with the general working of the software. As a means of producing a skeleton Petri net for a process description, it has its uses also, and in this way addresses the problem of finding Petri net components for process composition, fitting into techniques such as the client-server behaviour model (Birkinshaw and Croll, 1996). The tool itself does not dictate an interpretation of the specification, only the export interface does this.

6 REFERENCES

Birkinshaw, C. I. and Croll, P. R. (1995). Modelling the client-server behaviour of parallel real-time systems using Petri nets. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, vol 2, pages 339–348. ACM and IEEE Computer Society.

Birkinshaw, C. I. and Croll, P. R. (1996). A client-server approach to parallel software engineering. *Transputer Communications*, 3(1):33–40.

Duan, Z. (1994). A hybrid projection temporal logic for hybrid systems. In *Proceedings of the European Simulation Multiconference, Barcelona, Spain.*

Duan, Z. (1995). Hierarchic hybrid machines for refinements of hybrid systems. Technical Note 34, Mixed Mode Group, University of Sheffield.

Duan, Z. H., Holcombe, W., and Linkens, D. A. (1995). Specification of a soaking pit system in parallel hybrid machines. In *Euromicro 95*.

Fahrland, D. (1970). Combined discrete event continuous system simulation. *Simulation*, pp 61–72.

Ghezzi, C., Mandrioli, D., Morasca, S., and Pezze, M. (1991). A unified high-level Petri net formalism for time-critical systems. *IEEE Trans. Software Engineering*, 17(2):161–171.

Hoare, C. A. R. (1978). Communicating sequential processes. *Comms. of the ACM*, 21(8):666–677.

Jensen, K. (1990). Coloured Petri nets: A high level language for system design and analysis. In Rozenberg, G., editor, *Advances in Petri Nets 1990*, LNCS, pages 342–416. Springer-Verlag.

MetaSoft (1992). *Design/CPN Reference Manual*. Meta Software Corporation, 125 Cambridge Park Drive, Cambridge, Massachusetts, USA.

Yang, Y. Y., Linkens, D. A., and Banks, S. P. (1994). Extended coloured Petri nets and its application in mixed mode systems. In *Workshop of hybrid Systems and Autonomous Control. Cornell univ., NY.*

Yang, Y. Y., Linkens, D. A., and Mort, N. (1995). Modelling of a soaking pit/rolling mill process based on extended coloured Petri nets. *Control Eng. Practice*, 3(10):1359–1371.

INDEX OF CONTRIBUTORS

KEYWORD INDEX