System Specification for Applications with Dynamic Data Storage and Intensive Data Transfer

Julio Leao da Silva Jr., Chantal Ykman-Couvreur, Gjalt de Jong[*],
Bill Lin[+], Diederik Verkest, Francky Catthoor
IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
[*]Alcatel Telecom, F.Wellesplein 1, B-2018 Antwerp, Belgium
[+]University of California, San Diego, 9500 Gilman Drive, 92063-0407 La Jolla, USA

Abstract

Telecommunication systems are rapidly increasing in design complexity, while design time is shortening. Currently, each hardware component in these systems is specified, using hardware description languages. At this level, code is hard to read, modify/maintain, and reuse. In addition, simulation speed is a limiting factor for validation. To overcome these limitations we have developed the Matisse language and its underlying model. This paper presents Matisse, a concurrent object-oriented system specification language, well suited for telecom applications. The applicability of the Matisse language and the effectiveness of its underlying model is demonstrated by the results obtained for two industrial applications used in ATM networks. Moreover, Matisse allows design exploration of alternatives and dynamic memory management refinement which are incorporated in a methodology that bridges the gap between system specification and synthesis tools commercially available.

1 Introduction

Telecom network applications include system components for ATM based broadband networks, SONET and SDH based networks [32], mobile network infrastructures and interactive video-on-demand servers. These applications are among the fastest growing segments of the system industry today.

Modern telecommunication systems are rapidly increasing in design complexity to be able to support a wide variety of broadband multimedia services. In order to enable these services, elaborated network management is needed.

Implementation of telecom systems into a single chip hardware/software solution is no longer a problem from the technological viewpoint (e.g. [25]). Nowadays, the challenge is to design such electronic systems fast, efficiently, and first-time right. The realization of complex systems is becoming *design*

limited instead of *technology limited*.

Currently, there is an urgent need for (1) innovations in models and implementation-independent validation techniques at the system level and (2) coherent system design methodologies to bridge the gap from the system-level specification down to the low-level hardware and software development, using existing tool chains. This paper focuses on the first issue to be able to couple with a system design flow called the Matisse methodology [14, 16].

In the broad domain of telecommunication systems, we focus on protocol processing applications. These applications present several design challenges due to their characteristics and design constraints. They require manipulation of large amounts of irregular data that are dynamically created and destroyed at run time. They are characterized by tight interaction between control and data-flow behavior, intensive data transfers, and stringent real-time requirements. Due to area, power, performance and flexibility constraints, the physical to transport layers of these applications are usually (partly) realized in hardware or embedded software [48].

Protocol processing systems are extremely complex and they must be modeled, debugged, and simulated at a high-level of abstraction, before proceeding to implementation. A system specification language and its underlying model, well-suited to specify protocol processing systems must be independent from the final implementation, permit easy updating and efficient design exploration, allow successive refinements, and be easily retargetable to different embedded hardware/software realizations.

This paper presents two major contributions: (1) a system specification language, that supports high-level specification of protocol processing applications, and (2) its underlying model, that enables functional validation and is used as input to the exploration path.

The remainder of this paper is organized as follows. First, Section 2 gives the context and motivation for our work. Then, Section 3 presents relevant related work. Section 4 presents the main requirements that must be supported for our target domain, illustrated through an actual application example. Section 5 describes the Matisse language and model in a detailed way. Section 6 gives an overview of our hardware/software design flow. Section 7 presents specification and simulation results for two industrial applications. Finally, Section 8 presents the main conclusions.

2 Context and Motivation

Currently in industry, protocol-processing applications are designed starting from a document, written in the natural English language, which is often widely open for ambiguous interpretations. Then the system is partitioned into hardware and software components. These components are designed concurrently and independently from each other trying to minimize time to market.

Each software component is usually specified using SDL[42]. C or C++ code can then be automatically generated and compiled in machine instructions for the target processor. Run-time support is added for managing concurrency, by making use of functionalities such as task scheduling and interprocessor communication, offered by real-time microkernels [49] based on pre-defined architectures.

Each hardware component is manually specified as a finite state machine, using hardware description languages such as VHDL or Verilog [48]. Then a detailed implementation is produced by logic synthesis and technology mapping. The specification of each hardware component is done at the Register Transfer Level (RTL). However, RTL code is hard to read, modify/maintain and reuse. The code is already refined with detailed clock cycles, and specific architectural decisions are already fixed. Small changes at the system level often require very substantial changes in the hardware or software specifications of the components. Recently, the industry has started to use behavioral synthesis, but this only solves subproblems in the complete design flow, such as resource allocation and scheduling, register assignment and optimization, and data-path interconnection.

Since hardware and software components are realized independently from each other, this often introduces both specification and implementation mismatches, which are only detected at the final stages of the system design [23]. Moreover, system validation by executing these software and hardware specifications is time-consuming due to the vast amount of low-level details. Currently, the system integration and test phases take a major part of the complete system design [2].

Beyond that, due to the low level of entry in the currently used design flow, exploration of different alternatives is nearly impossible. Instead of exploring the design space to find the optimal solution, most design decisions are taken locally, based on previous designer experience. Exploring different HW/SW partitions and different data structures implementations would imply respecifying the component.

In future implementations of similar systems, functionalities of several processors will be integrated into a single VLSI chip, and some functionalities previously implemented in hardware will migrate to software, exploiting the increasing processing performance of emerging embedded microprocessor cores and enabling more flexible systems to be developed. Using the current design methodology, this is only feasible respecifying and reimplementing the complete component.

3 Related Work

Related work is divided in two parts: work done in the software community and work done in

the hardware community.

3.1 Software community

Distributed programming languages have been proposed for programming general-purpose multiprocessor systems or distributed networks of workstations [7, 10, 37, 43]. While their underlying models are related to our Matisse model, their implementation targets are different: they rely on elaborate run-time environments and are intended for pure software implementations. In contrast, our implementation target is intended for *optimized embedded* single-chip hardware/software realizations.

Several concurrent object-oriented (OO) languages extended from C++ exist. C** [34], Composites [9], Dome [3], DPC++ [22], Mentat [47], and QPC++ [4] are languages intended for specification of data level concurrency, while our goal is the specification of task level concurrency. CC++ [10], DC++ [8], DoPVM [28], and Presto [20] have either syntax extensions or library extensions in order to allow the specification of task level concurrency. However, they are based on large run-time environments intended for running on workstations, while our target is an embedded realization with minimum run-time support.

Some other extensions of C++, such as Charm++ [31], Concurrent C++ [21], and Panda [1], encapsulate tasks into communicating active objects. This type of languages seems to be the most appropriate to introduce concurrency into C++ without breaking the basic principles of object orientation, such as encapsulation. However, Charm++ allows various dynamic load-balancing strategies enlarging the run-time needed to support it. Concurrent C++ uses a centralized synchronization mechanism that makes it very hard to use the inheritance mechanism available in C++ for active objects. Panda allows different synchronization mechanisms such as monitors and semaphores. By enabling to mix synchronization mechanisms, both design task and verification task, e. g. deadlock detection, will be much more difficult to be implemented. Panda also allows dynamic migration of tasks from one processor to another enlarging the run-time needed to support it.

Our own proposal will be based on principles adopted in CC++ (see Section 5). However, we merge the active object concept with CC++ in an attempt to obtain a language that has the advantage of both approaches. We also introduce simplifications in CC++ in order to avoid a large run-time environment.

3.2 Hardware community

Several approaches exist focusing on modeling of embedded hardware/software.

The hierarchical FSM model is a powerful formalism for reactive control behaviors, but it

does not support well abstract data structures and OO features. Approaches intended for reactive control systems, such as Statemate [26] from iLogix, SpecCharts [40], Cosyma [19], and Esterel [27] lack support for data storage and transfers.

Heterogeneous design environments, like Ptolemy [6] and CoWare [5, 13], aim to provide an open environment to smoothly integrate different models of computation.

Most system-level research and CAD innovations today are focussed on Digital Signal Processing (DSP) applications (e.g. [6, 35, 36]). Commercial tools include SPW [46] from Alta/Cadence, COSSAP [12] from Synopsys, Monet [39] from Mentor Graphics, and DSP Station [18] from Frontier Design.

Protocol processing applications require manipulation of complex data structures that are often dynamically created and destroyed at run time, as opposed to the static signal flow present in DSP applications. Most DSP models are also not well-suited for control-oriented data processing behaviors found in protocol processing applications that heavily rely on tight interactions between control-flow algorithms and stored data structures. Due to many differences in nature between these application domains, system models should be domain-specific.

SDL has been used for specification of protocol processing applications in [45]. However, the crucial exploration of different dynamic memory management alternatives is unfeasible using this approach.

Related work on system synthesis for protocol processing applications has been dealing with different aspects, e.g., solving timing constraints [33] and memory allocation for minimizing memory interface traffic during HW/SW partitioning [30]. However, they have not dealt with languages or models for our target domain.

4 Specification Requirements

This section presents an actual protocol processing application used in telecom networks. This case study is used to illustrate the requirements to be supported by the Matisse language.

ATM [41] is a fast packet-switching transfer mode that supports high-speed integrated services by splitting all communication messages into equal 53-byte cells, called ATM cells. These cells can carry any kind of information, be it computer data, video, or voice. In addition, ATM networks are characterized by a connection-oriented mode of operation.

One representative case study is an actual industrial ATM based broadband network application developed by Alcatel. This application is a user transparent connectionless router called Alcatel Connectionless Transport Server (ACTS) [48] that provides the necessary functions for the direct provision and support of data communication between geographically distributed computers or between LANs over an ATM based broadband network.

In its current implementation, the ACTS consists of several boards, each one consisting of several processors and coprocessors, implemented as custom ASICs, and a programmable supervising microprocessor for executive control. A concrete example of one of those ASICs, named Segment Protocol Processor (SPP) [48], is used to demonstrate the requirements to be supported by the Matisse language.

The SPP can be described as a set of concurrent tasks that cooperate with each other through the shared data structures, as shown in Figure 1. More details about these tasks can be found in [48, 15]. These tasks have to be performed for each incoming frame, consisting of 4 ATM cells, at a frame rate of 622 Mbit/s. These tasks are combined, in order to satisfy design constraints, such as high memory bandwidth.

Figure 1: SPP task level diagram

The SPP stores and forwards user cells, performs a number of checks by itself, issues requests to other coprocessors to perform other checks, issues a request for routing and processes routing replies. The algorithms, implementing the SPP functionality, make use of ADTs, shown in Figure 2. The top of the figure shows a queue, where incoming user cells are buffered. Packet records are accessed through two keys: the local (LID) and multiplexing (MID) identifiers. A packet record contains various fields, such as the number of cells received so far, the time the first cell was received and a pointer to a list of routing records. In the target implementation, these ADTs are refined through the DMM refinement mentioned in Section 6.

Figure 2: SPP data organization

The shared data structures, such as the packet records and the FIFO, are used in the SPP as a communication mechanism between the six tasks. These tasks may operate concurrently, while respecting ordering dependencies. For instance, a cell is *first* stored in the FIFO. *Then* fields are updated in the respective packet record. *Afterwards*, depending on the type of cell, and after an observation time, an ISR request is generated.

Other ASICs used in the ACTS, such as the Packet Handler Processor and the Preventive Congestion Control processor and other network components may be described in a similar way, by means of a set of cooperative tasks operating on shared data structures.

In summary, the main characteristics of protocol processing applications are the following:

1. they manipulate large amounts of data, which are shared among tasks,
2. they perform a lot of data transfers,
3. they have coarse grain concurrency, defined by tasks, in which control constructs, such as *if-then-else*, *for* and *while loops*, are essential for capturing the algorithmic behavior of each task,
4. they have a small data arithmetic part,

5. they operate under real-time constraints, and
6. they target an embedded solution, in which power and area are crucial.

5 Matisse Language and Model

While an OO model is not necessary in principle, it has been proven successful to enable maintainability and code reuse in the software design community, and it also plays a central role in large-scale hardware/software system design. Object-oriented languages concentrate on the real-world entities identified in the application, which are tasks for accessing data in protocol processing applications.

Object-oriented languages support data abstraction, encapsulation, polymorphism, function overloading and inheritance, which are invaluable features in any large-scale development. With these abstraction facilities, implementation decisions and low-level specification details can be hidden or easily updated, allowing easy and fast design exploration. For instance, shared data structures may be initially specified as ADTs that will be refined later in the design flow.

Although the OO paradigm may incur design overhead, by restricting the system specification language and by automating the system design flow, these inefficiencies are minimized.

Since concurrent tasks to be executed on more than one processor need to be modeled, concurrent OO models are well suited to model protocol processing applications. Objects can encapsulate tasks as well as (shared) data structures. Remote procedure calls can encapsulate intertask communications. Consequently, a system specification language that supports a concurrent OO model is well suited for specifying protocol-processing applications.

The system specification language must also have the following characteristics: reflect the conceptual partitioning of the system, seen as a set of concurrent tasks for accessing data; be independent from the final implementation; permit easy updating and efficient design exploration; and be easily retargetable to different embedded hardware/software realizations. This contrasts with current system specification practices which are using VHDL for specifying the hardware processors, and C/C++ for specifying the software processors, or SDL and UML, which are too high-level for this target domain and do not provide the necessary hooks for efficient design realization and exploration.

From the previous requirements, we have decided to follow a *concurrent object-oriented* approach for the system specification of protocol processing applications. Therefore, the Matisse language is extended from the widely used OO programming language C++. We introduce minimal syntactic extensions to C++ to allow the description of concurrent tasks, communication and synchronization among them. Compatibility with C++ enables new users already familiar with C++ to be productive in a very short amount of time. In addition, existing debugging and compilation tools can be easily adapted for early

functional validation of the system specification. Finally, this enables us to leverage the wide corpus of existing software compilation and run-time support tools for our software implementation path. This is important since software implementation represents a substantial part of many of our target applications.

There are many concurrent OO programming languages extended from C++. All of them are intended for specifying systems consisting of concurrent programs, running on a network of workstations. Compilers for those languages generate C++ programs with calls to an elaborate run time Operating System (OS) designed for software processors only. Matisse is intended for specifying systems at the chip level instead. These systems consist of concurrent tasks, running on a mixture of embedded software and hardware processors. To be efficiently implementable in both hardware and software processors, the OS used by the specification language should offer only minimum support for task scheduling, interprocessor communication and synchronization.

More precisely, the Matisse language uses some of the high-level abstractions existing in Compositional C++ (CC++) [10]. CC++ is a concurrent OO language extended from C++ using only a few new keywords. Simplifications were introduced, taking into account the requirements introduced in Section 4 and also taking into account that systems specified with Matisse must be synthesized into a hardware/software codesign at the chip level. The OS requirements for CC++ (and similar languages discussed in Section 3) would be too costly to use in an embedded codesign context.

In protocol-processing applications, the user needs to specify concurrency only at the task level. In contrast to CC++ that allows the user to specify concurrency at all levels, from fine grain to task level concurrency, Matisse allows the specification of concurrency only at task level.

In CC++, both thread and local virtual memory space concepts are separated. To model tasks, Matisse allows to create active objects. Together, these objects encapsulate a local virtual memory space and define a default thread of control, that is initiated at the creation of the active object. Due to these restrictions, the complexity of the run time support can be reduced in a major way.

Similar to CC++, communication between tasks is abstracted, without explicit specification of communication channels and an RPC mechanism is used to implement it. In CC++, data may be remotely accessed directly. In Matisse, data inside an active object are remotely accessed only through a pointer to the active object itself. Due to the simplified communication mechanism, instead of providing two synchronization mechanisms as in CC++, Matisse only needs one synchronization mechanism.

Now the different concepts in Matisse are explained in more detail[1]. The concepts are illustrated using simplified code for the SPP driver application.

```
10  int main (int argc, char**argv) {
11     packet_record_mngr* global pr;  // shared data
12     cell_fifo_mngr* global cf;      // shared data
13     data_in* global di;             // task
14     data_out* global do;            // task
15
16     pr = activenew packet_record_mngr();
17     cf = activenew cell_fifo_mngr();
18     di = activenew data_in(pr,cf);
19     do = activenew data_out(pr,cf);
20  }

30  active class data_in {
31     cell_record* cell;
32     packet_record* packet;
33  public:
34     data_in ();
35     void body (packet_record_mngr *global pr,// packet_record_mngr,
36                cell_fifo_mngr *global cf,    // cell_fifo_mngr, and
37                input *global input){         // input are active classes
38                                              // pr, cf, and input are active objects
39         cell = input->get();         // get a cell from the input
40         switch (cell->type()) {
41            case BOM:                 // cell is Begin Of Message
42                packet = pr->alloc(); // create a new packet record
43                //SOME BEHAVIOR
44                pr->put(packet);      // store packet info into pr
45                cf->enqueue(cell);    // store cell in the fifo
46             case COM:                // cell is Continuation Of Message
47                packet = pr->get();   // use an existing packet record
48                // SOME BEHAVIOR      // retrieved from pr
49                pr->put(packet);      // store packet info into pr
50                cf->enqueue(cell);    // store cell in the fifo
51          }
52      };
53  };

60  active class packet_record_mngr {
61      packet_record *head, *tail;
62  public:
63      packet_record_mngr ();              // initialize head and tail
64      atomic packet_record* alloc ();     // create a new packet record
65      atomic packet_record* get ();       // get a packet record from the list
66      atomic void put (packet_record*);   // put a packet record in the list
67  };
```

5.1 Granularity of concurrency

The concurrency explicit in the specification of an application may vary from none to completely specified. According to the level of details present in the specification, three different levels are distinguished: abstract concurrency, explicit concurrency, and explicit decomposition into threads.

The definition of the concurrent tasks to be performed by the application is evident from the functionality, for dynamic data-dominated applications such as the SPP. Therefore, coarse grain

[1]Timing specification is still under investigation and it will not be presented in this paper.

concurrency in the specification is explicit in Matisse. However, finer grain concurrency (threads) is more difficult to be identified by the designer and is left implicit.

Each task is specified by means of an active object in Matisse (lines 30-52 in SPP code). Each active object specified in the application has its own thread of control. These active objects are concurrent. However, the internal concurrency in an object is implicit. This means that it is not necessary to specify the number of concurrent threads in an active object.

5.2 Level of concurrency

In case of the specification of concurrency using concurrent objects, a range of possibilities exists for specifying the concurrency inside an object. The concurrency existent internally to an object in the specification of an application may vary from *no concurrency* to *full concurrency*. According to the concurrency present inside an object, three different levels are distinguished: sequential, quasi-concurrent, and fully concurrent.

Fully concurrent objects are used in Matisse because they allow exploration of different alternative implementations. A concurrent object can be mapped to a more sequential or more concurrent implementation, according to a given cost function (e.g.: area, power). However, the most concurrent implementation of an active object is sometimes the only solution that enables to achieve the real-time requirements for dynamic data-dominated applications.

5.3 Concurrency type

To model different types of concurrency, such as data concurrency or task concurrency, different languages exist. In object-oriented languages, three possibilities for modeling concurrency are available: concurrent tasks, concurrent data, and active objects.

Dynamic data-dominated applications, such as the SPP, are best suited to be specified as a set of tasks operating on shared abstract data types. These tasks can be modeled as concurrent tasks or as active objects. The latter is best suited for introducing concurrency into an object-oriented language, in which only coarse grain parallelism is explicit. This is true because the latter maintains the encapsulation principle, while the former does not. Therefore, an active object model is used for modeling concurrency in Matisse.

Active objects and passive objects co-exist in a heterogeneous active object model. An active object is an instance of an active class. The specification of an active class is shown in lines 30-53 for the *active class data_in* in the SPP code. An active object has its own local virtual memory space and a default thread of control, specified in the *body* method of an active class. The default thread of control of active object is exemplified in lines 35-51. A passive object is used to model data, exactly as in C++.

In the *main* function in a Matisse specification, first active objects are created and then their bodies are initiated using the keyword *activenew*. In the SPP example (line 16-19), four concurrent active objects are shown: *pr*, *cf*, *di*, and *do*. Each one is an instance of a different active class in this example.

Active classes can inherit from base active classes, and the usual C++ protection mechanisms apply. So private data elements and member functions of an active class can be used only by the member functions of it. Public data elements and member functions constitute the interface to the active objects of the active class.

5.4 Abstract memory architecture model

The memory architecture model defines the way the memory is organized, for example, a centralized memory shared by all tasks or a distributed memory with each task owning its local memory. The term "abstract" is used because the memory architecture model defined here refers to the virtual memory and not to the actual memory.

An application can always be specified and implemented using different models for the abstract memory architecture. However, depending on the nature of the application, it may be best suited to be specified in one of the models.

Dynamic data-dominated applications, such as the SPP, are better specified by a number of shared abstract data types manipulated by a set of tasks. Some of these data types are shared by a set of tasks and some are local to one task. Therefore, an abstract distributed shared memory architecture model has been implemented in Matisse. This is illustrated by lines 11-14 in the SPP code.

5.5 Communication between objects

During communication, two (or more in case of broadcasted communication) partners are involved: a sender and a receiver. Here, the sender is the initiator of the communication independently of the direction the data is being communicated.

Different communication alternatives can be implemented, depending on whether the sender or receiver "blocks" waiting for the partner in the communication. Note that any combination sender/receiver with blocked/unblocked communication is possible, although some of them do not make sense.

Dynamic data-dominated applications are synchronous systems, in which a safe implementation of communication is needed. Therefore, blocked sender communication is used in Matisse, as illustrated in.

Communication among active objects is performed using *global* pointers. *Global* pointer

declarations are shown in lines 11-14, for the SPP example. The use of *global* pointers is shown in lines 39, 42, 44, and 45 for the SPP example. For instance, in line 42, the computation for the method *alloc* is executed in the active object *pr*. The communication proceeds in three stages:

1. first, the arguments of the function *alloc()* are packed into a message, communicated to the remote active object *pr*, unpacked and then the calling thread suspends execution,
2. next, a new thread is created in the remote active object to execute the called function, and
3. finally, upon termination of the remote function *alloc*, the function return value is transferred back to the calling thread which resumes execution.

Accessing data elements within an active object is regarded as local and hence cheap. A thread executing in an active object can access its data elements directly, by using C++ pointers. In contrast to data elements within an active object, active objects can be accessed by each other using *global* pointers. Except for their potentially higher cost of use, *global* pointers are used just like C++ pointers.

5.6 Synchronization mechanism

Due to concurrent computations, several accesses to data elements or member functions in an active object can occur simultaneously in dynamic data-dominated applications, such as the SPP. Therefore, these accesses have to be sequentialized. This is the only type of synchronization needed for this type of applications.

Several synchronization mechanisms can be used in concurrent object-oriented languages. Four classes of synchronization mechanisms are briefly compared. State-based synchronization mechanisms provide more than the minimal features and require also a much more complex compiler implementation. Centralized synchronization mechanisms have a complete sequential control flow of execution disabling more concurrent implementations. Explicit synchronization mechanisms (e.g. semaphores) are hard to use and error prone, requiring a lot of synchronization code. A monitor-based synchronization mechanism provides enough fine grain synchronization and enough synchronization capabilities, needed for dynamic data-dominated applications. Therefore, in Matisse, a monitor-based synchronization mechanism is implemented.

In particular, *atomic* functions are used to specify the synchronization mechanism. The specification of *atomic* functions is exemplified in lines 64-66 for the SPP. Whenever several threads call an atomic function, this function is executed the required number of times in a sequential order. Also, the execution of an atomic function never interleaves with the execution of another atomic function of the same active object.

Member functions may be declared *atomic* in both active and passive classes. However, in order to avoid deadlocks, some rules for defining atomic functions must be followed, such as:

- the body of an *atomic* function must terminate in a finite time, implying that it may not perform an RPC and that it may not call other atomic functions of its class;
- a *body* function running forever may not be declared *atomic*.

Using *atomic* functions instead of atomic objects helps the user to specify critical sections that must be as short as possible. In an object-oriented approach, each object (either active or passive) is responsible for its own protection. In Matisse, this is still valid, but deciding which member functions have to be declared *atomic* is currently left to the designer.

5.7 Language extension approach

Three alternatives for extending a sequential language exist: library-based, language syntax, and hybrid.

The advantage of library-based extensions is that the standard compiler for the sequential language is able to compile also the extensions contained in the libraries. Thus, there is no effort in building a compiler. Also, traditional tools (e.g. debugger) for the sequential language keep on working. However, in library-based extensions, it is not always possible to elegantly capture concepts due to syntactical limitations of the sequential language. Moreover, in library-based extensions global analysis is not feasible, since in the internal representation (after compilation) it is impossible to recover the information about concurrency.

For dynamic data-dominated applications, such as the SPP, global analysis of concurrency, communication, and abstract data types has to be performed. This analysis enables exploration of different implementation alternatives and selection of the best suited for a given cost function. Since global analysis is essential, a syntax-based extension has been implemented. In particular, the keywords *active*, *activenew*, *global*, and *atomic*, have been used in the SPP code.

The implementation of a hybrid solution is also possible. In this solution, first a library extension approach is adopted. Whenever global analysis is needed, a parser and semantical analyzer are used. Instead of parsing new keywords, as in the language extension approach, the classes in the libraries have to be parsed to create an internal representation keeping the concurrency information. This solution would bring the huge benefit of enabling use of available tools (compilers, debuggers, etc.) for C++. However, the parsing step still has to be done to enable global analysis. This is a subject for future implementation.

5.8 Processor mapping

Two alternatives exist for the mapping: explicit or implicit. In an explicit mapping, the designer must specify the mapping, while in an implicit mapping, the compiler defines the mapping.

An evident grouping of tasks to be mapped into a processor can easily be done for dynamic

data-dominated applications because the number of active objects is small. Therefore, the mapping of tasks into processors is explicit. For instance, the SPP has 11 active objects in total. Excluding interfaces and testbench, there are only six active objects to be mapped to processors.

5.9 Run-time functionality

Dynamic data-dominated applications require run-time support for task creation, task scheduling, concurrency, monitor synchronization, task communication and timing. The run-time library Tipsy[11] has been used to implement this functionality. The run-time support is incorporated into the Matisse underlying model by the Abstract Machine Generation step briefly described in Section 6.

6 Matisse Design Flow

The Matisse language (Section 5) is used as input to the system design flow [16]. The abstract machine model underlying the initial Matisse specification is used for functional validation and it facilitates design exploration due to its high-level of abstraction. The input to the design flow is a system together with its environment specified at the algorithmic level, using the Matisse language.

The system design flow is automated in order to avoid specification and implementation mismatches, to accommodate changes in the system specification, and to allow design exploration. The functional specification should be validated already at the system level, without executing both hardware and software low-level specifications, that are too time-consuming. This allows extensive exploration of design alternatives, such as refinement of Abstract Data Types (ADTs) into complex data structures (heaps, hash tables, trees and linked lists), refinement of the virtual memory management, and memory access optimization and memory synthesis. Below, we briefly present each step in the design flow. More details can be found in [16].

Abstract machine generation - The main goal is to obtain a library-based executable specification, that reflects the underlying Matisse model, called Abstract Machine (AM). The AM is suitable for simulation, exploration, and refinement of the system specification. The AM allows record access profiling for selecting optimized ADT implementations [51], intertask communication profiling for task concurrency management, and virtual memory access profiling for physical memory management [50].

Dynamic memory management (DMM) refinement - The Matisse language allows the designer to define sets of records as ADTs, without low-level specification details. Protocol processing applications are often characterized by algorithms that operate on large Data Structures (DSs), which are dynamically allocated. When implementing these applications on a chip,

efficient organization and implementation of sets of records [51] is crucial, and dynamic memory allocation [17] must be handled efficiently in terms of time and number of memory accesses. Hence, both the specification of sets of records and the memory management must be refined before synthesis.

Task concurrency management - The goal of task concurrency management is to meet the overall real-time requirements imposed to the application being designed, by making decisions at the task level without incorporating low-level operation details yet. Currently performed manually, a systematic methodology and the fully automation of this step are under investigation in our research.

Physical memory management for each processor - Typically, protocol-processing applications require large storage capacities and very high I/0 bandwidth to achieve the real-time requirements. Distributed memory architectures allow exploiting parallelism, thus alleviating memory access bottlenecks. However, as the required memory bandwidth increases, the cycle budget constraints available for each access individually become tighter. This happens because the number of addresses that have to be generated in parallel per processed data becomes higher, thus leading to an addressing overhead. This step aims at synthesizing area and power efficient distributed memory architectures and memory management units [38], [44], meeting the real-time requirements. In the Matisse design flow, techniques have been developed for pure HW processors first, where the memory organization is fully customized. Further investigation aims at extending the memory management steps to heterogeneous HW and SW processors with a partially predefined memory organization.

Finally, **software compilation** proceeds using traditional software compilers, **hardware synthesis** proceeds using High-Level Synthesis (HLS) tools and **interface synthesis** generates SW device drivers for each SW processor and VHDL specifications of the necessary HW blocks allowing communication between HW and SW processors. The interface synthesis is performed using the system integration toolbox *CoWare* [5] that also allows cosimulation of the synthesized system.

7 Results

This section presents the simulation results for two applications: the SPP, already used as an example in this paper, and the F4 [29], an Operation And Maintenance (OAM) component at the Virtual Path (VP) level of the ATM layer.

Simulation results include concurrent untimed, concurrent timed, and sequential simulation, for the SPP and F4. The exploration and synthesis results for the SPP can be found in [16] and for the F4 in [52].

7.1 Case study: SPP

The complete specification of the SPP consists of 2535 lines of Matisse code. The part of the Matisse libraries used for exploration for the SPP have 4376 lines of Matisse code from which 1062 have been used for implementation of the SPP.

The SPP application has been simulated for 1000 ATM cells. The elapsed time for simulating these cells is 1.5 seconds for the untimed concurrent specification written in Matisse.

After manually applied task concurrency management, a possible sequential specification (already scheduled) of the SPP has been obtained. This simulation has taken an elapsed simulation time of 0.8 seconds.

For an untimed concurrent specification, the simulation is about a factor of two slower than the sequential simulation, for the SPP. However, it is better to start from the concurrent specification from which we can derive different sequential specifications. This enables design exploration of different alternatives.

A concurrent timed simulation has not been done for the SPP because the relative timing of tasks is not part of the SPP specification.

7.2 Case study: F4

The complete specification of the F4 consists of 3559 lines of Matisse code. The part of Matisse libraries used for exploration for the F4 have 3998 lines of Matisse code, from which 894 have been used for the implementation of the F4.

The F4 performs functions such as fault management, performance monitoring, fault localization, and activation/deactivation of VP tasks. It is present on each physical link connected to the ATM switch. To perform its functions, the F4 block deals with specially marked ATM cells, called OAM cells. These cells are distinguished from user cells by dedicated values for the VP identifiers. All functions have to be performed for each incoming ATM cell in 2734 ns.

The F4 application has been simulated for 1000 cycles (representing 2000 cells processed). The elapsed time for simulating the processing of these cells is 15 seconds for the timed concurrent specification written in Matisse.

After manual task concurrency management, a possible timed sequential specification (already scheduled) of the F4 has been obtained. This simulation has taken an elapsed time of 15 seconds. For the F4 application, the simulation of the timed concurrent specification is as fast as the sequential specification.

A concurrent untimed simulation has not been done for the F4 because the relative timing of tasks is part of the F4 specification. The functional validation and the timing validation have been

done together.

8 Conclusion

None of the existing languages and models has been designed for the specification of hard real-time dynamic data-dominated applications, which should be realized in an embedded solution, except for the proposed Matisse language and its underlying model.

In contrast to other languages and models, the Matisse language and model combine the following characteristics, crucial for our target domain:

- they meet the requirements needed for concurrency, communication, synchronization;
- they target an embedded solution;
- they enable fast and effective functional validation;
- they are usable as input for exploration and refinement of the system specification;
- they allow profiling for dynamic memory management, task concurrency management, and physical memory management.

Currently, we are investigating how to include timing constraints in the system specification and support them through the system design flow to be able to handle other types of applications in the network component domain.

In the near future, we want to show the applicability of our concurrent OO approach on other actual telecom and multimedia (e.g. MPEG4) applications which also incorporate dynamically created data types.

Acknowledgments

We would like to thank K. Croes, E. Umans, J. Cockx, and P. Six for numerous insightful discussions. We would like to thank A. Hemani, B. Svantesson, and P. Ellervee from KTH (Sweden) for providing us the specification of the F4 application. This research has been partly funded by the Flemish IWT and Alcatel in the HASTEC project, by the Esprit project MEDIA (No.21929), and by a Brazilian Government Fellowship (CAPES).

References

[1] H. Assenmacher, "PANDA - Supporting Distributed Programming in C++", *Proc. of European Conference on Object Oriented Programming*, pp. 361-383, 1993.

[2] B. Bailey and S. Leef, "Making the shift toward integrated systems design", *Electronic Design*, Vol. 44, No.14, pp. 86-92, 1996.

[3] A. Beguelin et al., "Dome: Distributed Object Migration Environment",. Technical Report CMU-CS-94-153, School of CS, Carnegie Mellon University, May 1994.

[4] D. Boles, "Parallel Object-Oriented Programming with QPC++", *Structured Programming*, Vol. 14, pp. 158-172, 1993.

[5] I. Bolsens et al., "Hardware-software codesign of digital telecommunication systems", *IEEE Proceedings*, April 1997.

[6] J. Buck et al. "Ptolemy: A framework for simulating and prototyping heterogeneous systems", Technical report, University of California, Berkeley, August 1992.

[7] D. Caromel, "Toward a method of object-oriented concurrent programming", *Communications of the ACM*, September 1993.

[8] H. Carr et al., "Distributed C++*ACM Sigplan Notices*, Vol. 28, No.1, Jan. 1993.

[9] M. Carrol and L. Pollock, "Composites: Tree for Data Parallel Programming", *Proc. of the Intnl. Conference on*

Computer Languages, pp. 43-54, Toulouse, France, May 1994.
[10] K. M. Chandy and C. Kesselman, "CC++: A declarative concurrent object-oriented programming notation", MIT Press, 1993.
[11] J.Cockx, "Concepts, methods, and C++ classes for executable system modeling", Internal report, IMEC, Leuven, Belgium, March 1998.
[12] http://www.synopsys.com/products/dsp/cossap_br.html
[13] CoWare, 2900 Gordon Av., Santa Clara, CA 95051, USA and Kapeldreef 60, B-3001 Heverlee, Belgium, http://www.coware.com/.
[14] J. L. da Silva Jr., C. Ykman-Couvreur, G. de Jong, B. Lin and H. De Man, "A system design methodology for telecommunication network applications", *The Seventh Great Lakes Symp. on VLSI*, 1997.
[15] J. L. da Silva Jr., C. Ykman-Couvreur and G. de Jong, "MATISSE: a Concurrent and Object-oriented System Specification Language", *Proc. of the Intnl. Conference on VLSI (IFIP)*, August 26-29, 1997.
[16] J. L. da Silva Jr, C. Ykman-Couvreur, M. Miranda, K. Croes, S. Wuytack, G. de Jong, F. Catthoor, D. Verkest, P. Six and H. De Man, "Efficient System Exploration and Synthesis of Applications with Dynamic Data Storage and Intensive Data Transfer", *Proc. 35th ACM/IEEE Design Automation Conf.*, San Francisco CA, June 1998.
[17] J. L. da Silva Jr, F. Catthoor, D. Verkest and H. De Man, "Power exploration for dynamic data types through virtual memory management refinement", *Proc. of Intnl. Symp. on Low Power Design*, Monterey CA, Aug. 1998.
[18] http://www.frontierd.com/dspstation.htm
[19] R. Ernst, J. Henkel, T. Benner, "Hardware-software cosynthesis for microcontrollers", *IEEE Design and Test of Computers*, Vol. 10, No.4, pp. 64-75, Dec. 1993.
[20] J. Faust and H. Levy, "The Performance of an Object-Oriented Threads Package", In *Proc. of European Conference on Object Oriented Programming/OOPSLA*, pp. 278-289, Oct. 1990.
[21] N. Gehani and W. Roome, "Concurrent C++: Concurrent Programming with Class(es)",*Software - Practice and Experience*, Vol. 18, No.12, pp. 1157-1177, Dec. 1988.
[22] S. Gilbert, "A MasPar implementation of Data Parallel C++", Technical Report, London Parallel Application Center (LPAC), Sep. 1992.
[23] R. Glover, "EDA Vendors Advance Toward System-Level Design", *Electronic Design*, pp. 77-80, July 8, 1996.
[24] R. Gupta, G. De Micheli, "Hardware-software cosynthesis for digital systems", *IEEE Design and Test of Computers*, Vol.10, No.3, pp. 29-41, Sep. 1993.
[25] B. Gyselinckx, L. Rijnders, M. Engels and I. Bolsens, "A 4*2.5 Mchip/s Direct Sequence Spread Spectrum Receiver ASIC with Digital IF and Integrated ARM6 Core", *Proceedings of the IEEE Custom Integrated Circuits Conference*, Santa Clara, CA, pp. 461-464, May, 1997.
[26] D. Harel, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering*, Vol. 16, No.4, April 1990.
[27] N. Halbwachs, "Synchronous Programming of Reactive Systems", Kluwer Academic Publishers, Dordrecht, the Netherlands, 1993.
[28] C. Hartley and V. Sunderam, "Concurrent Programming with Shared Objects in Networked Environments", *Intnl. Parallel Processing Symp.*, April 1993.
[29] A. Hemani, et al, "Design of Operation and Maintenance Part of the ATM Protocol!", *Journal on Communications*, Hungarian Scientific Society for Telecommunications, special issue on ATM networks, 1995.
[30] A. Jantsch, et al., "Hardware/software partitioning and minimizing memory interface traffic", *Proceedings of the EuroDAC*, 1994.
[31] L. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++",*ACM Sigplan Notes*, Vol. 28, No.10, Oct. 1993.
[32] B. Kumar and J. Ranade, "Broadband Communications, A Professional's Guide to ATM, Frame Relay, SMDS, SONET and B-ISDN", McGraw-Hill Series on Computer Communications, 1994.
[33] K. Kuchcinski, et al., "Embedded System Synthesis by Timing Constraint Solving", *Proceedings of the Intnl. Symp. on System Synthesis*, Sep. 1997.
[34] J. Larus et al., "C**: a Large-Grain, Object-Oriented, Data-Parallel Programming Language", Technical Report 1126, CS Department, University of Wisconsin-Madison, Nov. 1992.
[35] R. Lauwereins et al., "Grape-II: A system-level prototyping environment for DSP applications", *IEEE Computer*, pp. 35-43, February 1995.
[36] H. De Man et al., "Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms", *IEEE Proceedings*, Vol. 72, No.2, pp. 319-335, February 1990.
[37] B. Meyer, "Systematic concurrent object-oriented programming", *Communications of the ACM*, September 1993.
[38] M. Miranda, F. Catthoor, M. Janssen and H. De Man, "ADOPT: Efficient Hardware Address Generation in Distributed Memory Architectures", *Proc. 9th ACM/IEEE Intnl. Symp. on System-Level Synthesis*, La Jolla CA, pp. 20-25, Nov. 1996.
[39] http://www.mentorg.com/monet/index.html
[40] S. Narayan et al., "System specification and synthesis with the SpecCharts language", In *IEEE Intnl. Conference on Computer Aided Design*, pp. 266-271, Nov. 1991.
[41] M. De Prycker, "Asynchronous Transfer Mode, Solution for Broadband ISDN", Ellis Horwood, 1991.

[42] R. Saracco and P. Tilanus, "CCITT SDL: An overview of the language and its applications", *Computer networks and ISDN Systems, Special Issue on CCITT SDL*, Vol. 13, No.2, pp. 65-74, 1987.
[43] B. Selic et al., "Real-Time Object-Oriented Modeling", Wiley, 1994.
[44] P. Slock, S. Wuytack, F. Catthoor, and G. de Jong, "Fast and Extensive System-Level Memory Exploration for ATM Applications", *Proceedings 10th ACM/IEEE Intnl. Symp. on System-Level Synthesis*, Antwerp, Belgium, Sep. 1997.
[45] B. Svantesson, Sh. Kumar, and A. Hemani, "A methodology and algorithms for efficient interprocess communication synthesis from system descriptions in SDL", *International Conference on VLSI Design*, Chennai, India, January 1998.
[46] http://www.cadence.com/alta/products/spwhds_dat.html
[47] The Mentat research group, "Mentat 2.8 Tutorial", Technical Report, University of Virginia, June 1994.
[48] Y. Therasse, G. Petit, and M. Delvaux, "VLSI architecture of a SDMS/ATM router", *Annales des Telecommunications*, 48(3-4), 1993.
[49] E. Verhulst, "Virtuoso: Providing submicrosecond context switching on DSPs with a dedicated nano kernel", *Intnl. Conference on Signal Processing Applications and Technology, Santa Clara*, Sep. 1993.
[50] S. Wuytack et al., "Flow graph balancing for minimizing the required memory bandwidth", In *Proceedings of the Intnl. Symposium on System Synthesis*, pp. 127-132, 1996.
[51] S. Wuytack et al., "Transforming set data types to power optimal data structures", *IEEE Transactions on Computer-Aided Design*, Vol. 15, No.6, pp. 619-628, June 1996.
[52] C. Ykman, et. al., "Stepwise Exploration and System Synthesis from SDL of an Operation and Maintenance Component in ATM switches", (submitted) *Design Automation Conf.*, June 1999.

Figure 1: SPP task level diagram

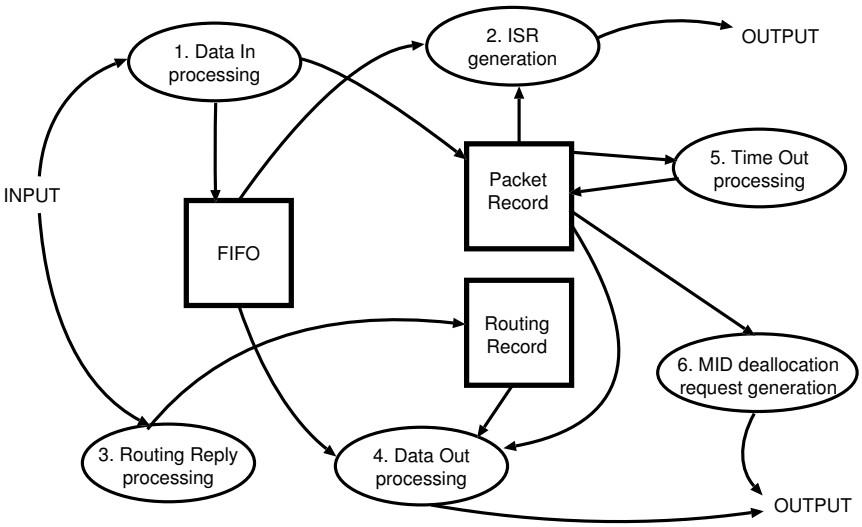

Figure 2: SPP data organization