IMPLEMENTING COMPLIANCE IN PROCESS-CENTERED SOFTWARE ENGINEERING ENVIRONMENTS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

July 2004

By
Wykeen Seet
Department of Computer Science

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Contents

Abstract 10

Declaration 11

Copyright 12

Acknowledgements 13

1 Introduction 14

1.1 Process Modeling 15

1.1.1 Software Engineering 15

1.1.2 Definitions 17

1.1.3 The Nature of Process Modelling 19

1.1.4 Process-Centered Software Engineering Environments . . . 20

1.1.5 Some Current PSEEs 21

1.1.6 Some Characteristics of PSEEs 22

1.2 Evolution(Softness) in Process Modelling 22

1.2.1 Some Characteristics of Evolution 23

1.2.2 Managing Evolution Complexity 24

1.2.3 Meta-Process 25

1.2.4 Hierarchical Structures 25

1.3 Formality(Hardness) in Process Modelling 26

1.4 Hypothesis 27

1.5 Research Methodology 29

1.6 Research Contributions 31

1.7 Thesis Structure 32

1.8 Summary 34

2 The Compliant Systems Architecture(CSA) 35
2.1 Introduction . . . 35
2.2 The Compliant Systems Architecture Approach . . . 36
2.2.1 Generic Compliance . . . 38
2.2.2 Definition of Compliance in Other Research Areas . . . 40
2.3 Determining Compliance . . . 44
2.3.1 Additional Properties of Compliance . . . 47
2.4 A Definition of Compliance for PSEEs . . . 48
2.4.1 Policy requirements of a PSEE . . . 48
2.5 Description of CSA Tools . . . 49
2.5.1 ArenaOS . . . 50
2.5.2 ProcessBase and PBAM . . . 50
2.5.3 The HyperCode System . . . 51
2.6 Summary . . . 53

3 The π-SPACE Language 55
3.1 Introduction . . . 55
3.2 Overview . . . 56
3.3 π-SPACE . . . 57
3.3.1 The π-calculus as used in π-SPACE . . . 57
3.3.2 π-SPACE types . . . 59
3.3.3 Aggregates . . . 60
3.3.4 Operations on Channels . . . 63
3.3.5 Operations on the *Aggregates* type . . . 64
3.4 Support for Dynamic Evolution . . . 65
3.5 Summary . . . 67

4 Language Compliance 68
4.1 Introduction . . . 68
4.2 Design of the enactable π-SPACE Language . . . 69
4.2.1 Recursive Descent Compiling . . . 69
4.2.2 Lexical Refinements . . . 70
4.2.3 Syntactic Refinements . . . 71
4.2.4 Semantic Refinements . . . 78
4.2.5 Code Generation . . . 81
4.2.6 Enaction Issues . . . 83

4.3 Language Compliance 85
4.3.1 Compliance in π-SPACE 85
4.4 Criteria for Language Compliance 86
4.5 Summary 88

5 Virtual Machine Compliance **89**
5.1 Introduction 89
5.2 VM Design Approaches 89
5.2.1 A Definition of Virtual Machine 89
5.2.2 Conventional VMs 91
5.3 Compliance in VM Construction 94
5.3.1 Support for Compliance in the PBAM 94
5.3.2 Comparisons of PBAM with conventional VMs 96
5.4 Design of πPVM 96
5.4.1 Architecture 97
5.4.2 Mechanisms to support Passive Compliance 97
5.4.3 Mechanisms to support Dynamic Compliance 108
5.5 Criteria for VM Compliance 110
5.6 Model for determining VM Compliance 111
5.7 Summary 113

6 Application Compliance **117**
6.1 Introduction 117
6.2 A π-SPACE HyperCode System 118
6.2.1 Preliminaries 118
6.2.2 Conceptual Model 118
6.2.3 Physical Model 123
6.3 Determining the Compliance of the π-SPACE HyperCode System 126
6.3.1 Conceptual Model 126
6.3.2 Physical Model 128
6.4 The Towers Software Process Framework 129
6.5 Integration of HCS and Towers 133
6.5.1 Simplifications of Towers 133
6.5.2 WebServices 134
6.5.3 Determining the Compliance of Integrated HCA and Towers 134
6.6 Criteria for Application Compliance 136

6.6.1 Static Compliance 136
6.6.2 Dynamic Compliance 137
6.7 Summary 137

7 Evaluation of Compliance **139**
7.1 Introduction 139
7.2 The Evaluation Approach 139
7.2.1 Objectives 140
7.2.2 Process 141
7.3 Evaluation of compliance on integrated layers 142
7.3.1 Integrating the compliant layers 143
7.3.2 Determining Compliance 145
7.3.3 Summary of findings 145
7.4 Evaluation of a csa for a PSEE 146
7.4.1 A Sample Application Process Model: The Writer Checker(W-C) Model 147
7.4.2 Comparisons of Evolution Modeling and Support 149
7.5 Non-Compliance 153
7.5.1 Communication model 153
7.5.2 Thread Control model 154
7.6 Summary 156

8 Discussion and Future Work **158**
8.1 Introduction 158
8.2 Compliance Model on the PSEE 158
8.2.1 Determination of a csa 158
8.2.2 A model of Active Compliance 158
8.3 HyperCode and the π-SPACE language 159
8.4 Compliance as a method for construction 160
8.5 The CSA Tools 160
8.6 Future Work 162
8.6.1 Language Compliance 162
8.6.2 Compliance in Hardware 163
8.6.3 Mechanisms and Policies as processes 163
8.6.4 From Determination to Measurement 163
8.6.5 Derived Work 164

8.7 Summary 164

Bibliography **166**

A Enactable π-SPACE **177**

A.1 Introduction 177

A.2 Reserved Words 177

A.3 Grammar in EBNF 178

A.4 Code Generation Rules 184

B The Tower Model **193**

B.1 Towers in π-SPACE 193

B.2 HDev Node Component 193

B.3 Specify Method 195

B.4 Verify Method 195

B.5 Node 196

List of Tables

4.1 Annotations in enactable π-SPACE 75
4.2 Local variables in Enactable π-SPACE 76
4.3 Differences of the textual representation of communication channel operations between specification and enactable π-SPACE 77
4.4 Differences of the behaviour definitions for components and connectors between specification and enactable π-SPACE 78
4.5 An example code generation rule that shows the type definition and Instance generator in ProcessBase of a π-SPACE component . 83
4.6 Code generation rules for Operation parameters 84

6.1 π-SPACE HyperCode Operations and their Domain Operations . 120
6.2 Effects of the Explode operation on π-SPACE Hyperlink types . . 122
6.3 Refinement of the Original Tower operations 135

List of Figures

1.1 Compliance and environment flexibility 29

2.1 The CSA model Layers of Policies Mechanisms and Binding . . . 37
2.2 Process Feedback Control Model 44
2.3 A model of the csa showing the required components of a compliant system 47
2.4 Conceptual model of a HyperCode System 52

4.1 Language Compliance, Compiler and Language 87

5.1 The Architecture of the πPVM 98
5.2 Types in Communication Control 100
5.3 π-SPACE types and their associated representations in ProcessBase 103
5.4 Global Control Structures in the πPVM 114
5.5 Physical Architecture of libraries in ProcessBase 115
5.6 Physical and compliant models of the πPVM 116

6.1 The Conceptual Model of the π-SPACE HyperCode System . . . 119
6.2 The Architecture of the π-SPACE HyperCode System 123
6.3 A screenshot of the HCA showing the added π-SPACE button . . 124
6.4 The customisations made for the π-SPACE HCS 125
6.5 Conceptual Model of the π-SPACE HyperCode System as a Compliant Systems Architecture 127
6.6 Physical Model of the π-SPACE HyperCode System as a Compliant Systems Architecture 129
6.7 The Tower Model which consists of the Node(including Operations) and the P^2E Metaprocess 132
6.8 The result of applying the csa model on the Towers Software Framework 134

6.9 The resultant architecture of integrating the Towers Node with the HCS 135
6.10 The resultant model from applying the csa determination model on the integrated Towers and HyperCode System 137

7.1 The resultant model that is derived from the integration of the Application, Language and VM Layers 146
7.2 The W-C model and some illustrations of W-C model evolution . 149
7.3 Construction of the W-C model using the HyperCode System . . 152
7.4 A Compliant Architecture view of the Thread Scheduler mechanism 155
7.5 A Final Compliant Systems Architecture model 157

8.1 A model of Active Compliance 159
8.2 The CSA model of extending mechanisms 161

Abstract

Software engineering requires an immense effort in finding the balance between the rigour required for defining and constructing a concrete end product and the flexibility required for ensuring that the 'final' end product can evolve in response to changing needs. A Process-Centered Software Engineering Environment (PSEE) requires a corresponding balance between the requirement to model and enact well-known processes and to be configurable to support informal and less well-defined processes. This makes a PSEE a particularly demanding software application in terms of the requirements for sufficient rigour for specifying processes and their predicted evolution, and sufficient flexibility for handling unpredicted evolution. The notion of a compliant systems architecture was based on the observation that for many large, long-lived applications there is a mismatch between the application requirements and the facilities required by the languages and operating systems. In a compliant system the underlying system would be flexible and configured to the needs of the application. If the application needs evolved then so would the system architecture.

This thesis consolidates existing work on compliant systems architecture. It provides a concrete definition of compliance, describing the properties that distinguish between compliant and non-compliant systems. This is based on a exemplar implementation that covers the complete range from the application to the virtual machine. This exemplar implementation is that of a PSEE that enacts a formal architecture description language (ADL). The flexibility of the compliant system is exploited in configuring the architecture of the process which generates the architecture of the product being produced. This facilitates the re-configuration of the product architecture when the product requirements evolve due to changes in the operating environment. This experience leads to a set of concepts and guidelines for constructing a compliant PSEE for enacting a formal ADL.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

Firstly, I would like to thank Prof Brian Warboys, my supervisor, who has been extremely understanding and supportive during this period of intense development and challenge of my life. Without his guidance, mentorship and reassuring support, this endeavour would not have been possible.

I would also like to thank my colleagues and friends, Mark Greenwood, Ian Robertson and Bob Snowdon who have been ever patient, understanding and during most times had to put up with the interruptions to their work from my constant questioning.

My parents, Chee Hong Seet and Yit Hoe Chan, without whom I will not be able to play this little role in the world. Thank you for bringing me up to be the person I have become so far. I hope my little achievement so far have at least made you that little bit more proud that I have been useful to society. My thanks to my brother, Wymen Seet, and sister, Elaine Seet, who have been providing constant support and encouragement over the course of my research. You have made being away from home alone all that much bearable.

I would also like to thank Aoy(Ms Suphasinee Limpanuphap), who has been giving me all the support during the tough times even though she had to work on her thesis as well. Thank you for the meals that nourishes me and your cheerfulness that keeps my spirits up so that I can keep working effectively in order to complete this thesis.

And lastly but certainly not the least, to my close-knit group of friends, Kuveshni Govender, David Smyth Boyle and Efrosini Deligianni for always being there for me through both the good and bad times.

Chapter 1

Introduction

This thesis is concerned with the use of the Compliance Systems Architecture(CSA) approach for constructing evolvable systems. It demonstrates that such an approach provides better support for evolvable process support systems than those that were constructed using traditional software construction approaches. The chapter establishes the scope of the research by first providing an overview of the area of *process modelling*. This is achieved by introducing the fundamental concepts, definitions and key problems that were identified from a survey of the literature. The key role of a Process-Centered Software Engineering Environments(PSEE), for enacting process models, is to support real-world processes. This approach to improving the software development process and thus also the subsequent software artefacts generated from the process will then be introduced.

Much was promised by the use of PSEEs in improving the software process and consequently the development of software but their utility in real-world software development was far from significant. Some illustrations of current PSEEs will be provided with emphasis on understanding the reasons that resulted in them not being widely adopted. The problem area will then be derived and a proposed model that is based on a definition of compliance will be described which forms the premise for this investigation.

The research methodology will provide details on the research objectives, approach and expected contributions from this investigation. An overview of the chapters documents the research approach and the findings completed for the thesis.

1.1 Process Modeling

This work concerns the study of process modeling in and, in particular its use to improve, the production of software artefacts. As such, the problems faced by process modeling are similiar to those that are faced by Software Engineering in general.

1.1.1 Software Engineering

No exposition on the *software process* field would be complete without a mention of the *software crisis* which incidentally led to the need for a 'Software Engineering'[60] field. The awareness of the software crisis and its detrimental effects spurred the interest to understand the main causes that led to the crisis and to provide better solutions in order to resolve the crisis.

Some initial approaches focused on refining the toolset that were already available where each was constructed independently to solve specific problems of software construction. This approach has been reasonably successful in making it easier for software developers to construct software that is more complex and larger than previously possible. Some of the resultant tools and technologies from the work done in this area are compilers, interpreters, operating systems, virtual machines and code editors. This focus on the tools and technologies however, in general, only addressed the coding phase of software development.

Another approach makes the assumption that the crisis was caused by the lack of understanding of the problem domain which was exacerbated by the abstract nature of software. An excellent image of how the abstract nature of software could lead to difficulties in developing software can be found in the seminal article by Brooks[14]. The illustration of a group of animals in a tar pit trying to escape it but getting more trapped as they sink into the solidifying tar suggests that tools by themselves are not sufficient to avoid or to tread on the tar pit. In addition, developers need to know how to mold the tar such that they can tread on it when required. In order to achieve this, the properties of software need to be understood and specified with sufficient rigour.

The issue was mainly identified to be due to the lack of notations to aid in the analysis and the specification of the problem domain and the proposed software artefact solution. The solution provided by this approach resulted in the introduction of new notations with well-defined syntax and semantics. In a way,

the formality provided a grounding that allowed the abstract nature of software to be made sufficiently concrete such that it could be understood and transmitted to different developers. Providing rigour to the software artefact and the problem domain provided a context that allows the understanding of the properties and characteristics of the specified software. Some notations that were introduced and used with reasonable success were the Vienna Development Method(VDM)[35], the Z notation[34], B[102], Communicating Sequence Processes(CSP)[32] and π-calculus[54]. However, most formal notations has so far been only applied to domains which are either very simple cases or mission critical.

The work undertaken in previous approaches could be viewed as refining the available primitive mechanisms which addressed the spectrum of problems that ranged from the more concrete approach of software coding to the abstractness of specifying the problem domain and the software artefact. The knowledge gained from the previous undertaking laid the foundation in order to improve the development of software. Early indications of this approach can be seen in the development of notations that are more focused on their utility to provide a guide to aid the development of software rather than on the rigour of the notation. The result are semi or non-formal notations and methods such as the Structured Analysis and Design Method(SADM)[24, 104], Object-Oriented Analsis and Design Method(OOADM)[11] and more recently the Unified Modeling Language(UML)[12] which essentially is a unified notation for OOADM.

The use of a *method* or a combination of them provides an approach for understanding the nature of software development itself. The realisation is that each method provides a guideline or rule of thumb for a usage *pattern* of the tools, technologies and notations. Further studies reveals that some organisation of these usage patterns results in the production of better software whereas some patterns produced software that were less than desirable. These patterns are now generally accepted to be a *process*.

In a more recent survey of the state of software engineering, Wasserman[100] listed eight key ideas ideas proposing a foundation of concepts, two of which, the lifecycle and process, tools and integrated environments are directly relevant for advancing the software engineering field and which he claims has already been addressed by current studies in the software process field. The view is that even though many high quality software were produced without an organized and disciplined software process, these were exceptions rather than the norm as

the importance of a software process to the success of a software development project increases as the size of a team increases. He also cautioned that this in no way invalidates that in smaller software projects, the software process is not important. It can only imply that the level of support provided by tools in smaller development teams in order to support the software process need not be as sophisticated and thus can be managed and performed by an individual. This suggests that a software process must be tailored to specific situations and needs.

An indirect result of the focus in process tailoring prompted the introduction of lightweight processes whereby the processes are described more informally as a set of guidelines and rule of thumb that are known to improve the construction of software. The recent interests in Extreme programming[8] and agile processes, which have generated a few conferences, seems to have struck a chord with those who were disillusion with the original promised silver bullet[14] solution to the software crisis.

The study of the *software process* is thus a significant area of research which can contribute towards the goal of achieving the holy grail of 'Software Engineering'. This, perhaps, was the reason why the Ninth International Conference on Software Engineering in 1987 had a significant number of papers on the software process where the seminal paper by Osterweil[62] described how the issues faced in traditional programming could relate to that of process programming.

A significant amount of work has been done since then. Warboys[97] provided some early reflections of the significant role that process modeling can attain rather than being limited to within the traditional domain of software engineering.

Curtis[21] added that the modeling of processes provides an overview of the current state of an organisation's real-world process model. The process model thus reveals what the creator of the process believes is vital in understanding or predicting the phenomena modelled. This understanding of process models thus offers another approach for tackling the 'software crisis' problem. The following section provides some definitions of a process and its key attributes.

1.1.2 Definitions

A software process[26, 27] is described as consisting of *process steps*, the purpose of which is to produce artefacts. Each process step is then further defined as a either a *task* if it is managed or an *activity* if the step is unmanaged. A managed process step is defined as one whereby resources are allocated for it, a schedule is

attached to the process step, assigned to an agent, and its progress is monitored against expectations.

Process steps are themselves performed by *agents*. These agents could either be human or machine.

Another definition of a software process as provided by Ould[64], described a process by not providing a direct definition of what a process is but rather by describing the key features of a process. According to Ould, processes are:-

1. purposeful activity
2. carried out collaboratively by a group
3. often crosses functional boundaries
4. invariably driven by the outside world

These attributes point to a process being driven by human goals and activities.

Some other relevant characteristics that a process model and its formalism should consider in order to model a real-world process can be found in Conradi[19]. The characteristics are listed as follows:-

1. Modularisation
2. Abstraction
3. Formalisation
4. Understandability
5. Clarity and Orthogonality
6. Evolution and Customisation
7. Monitoring and feedback

Most of these characteristics are focused on managing the complexity of the notations and the resultant model that has been specified. The charateristics numbered 1 - 5 are not dissimiliar to the characteristics which a typical software artefact should possess. However, the characteristics of Evolution and Customisation(6), and Monitoring and feedback(7) provide an insight into the additional complexity faced by the modeling of software processes due to its inherent need to evolve in order for the process model to continue to be useful for supporting the real-world process.

1.1.3 The Nature of Process Modelling

Osterweil[62] in his seminal paper titled 'Software processes are software too', provided a key thought in that the problems that were faced by modeling software process are the same ones that software developers have been facing in constructing software systems. The term Process Programming' was arguably made popular in this paper. Yet, Lehman[39] in his response to Osterweil, though praising the contribution as being useful to certain areas, cautioned that the algorithmic biased view of formalising a process does not address the problems that are caused by the informal needs of a process.

This notion is reiterated again later by Osterweil[63] where he provided a more refined view described in his original paper[62]. He argues that the original view of process programming was not what he would describ as 'process coding'. The realisation is that process coding implies a one way process where the process is elicited from observing the real-world process. However, this assumption is now realised to be rather myiopic in that process code also affects the functioning of the real-world process. This results in the need to have active models[84] that are constantly being updated to reflect real-world changes.

Ould[64] provided a set of 'Laws of Process Modeling' which generally concern business processes but, by their very nature, software processes are not just about the technical aspects of providing support for tools. They are also the support technology that works as the 'glue' to support tools and humans that work together to achieve a common, objective even if tools are not cogniscent of the objective.

Cugola[20] provided a more current update of the problems that are still being faced by practitioners and researchers in this field. He noted that even though substantial progress was made in the field, some key challenges remain. Process programming and the use of a PSEE were highlighted to be key challenges that require more research. The *rigidity* of the environment was noted as a key problem that prevented the PSEE from supporting, what was described as, a form of process evolution that deviates from that built-in as part of the environment.

1.1.4 Process-Centered Software Engineering Environments

It can be argued that the concept of a Process-Centered Software Engineering Environment(PSEE) is a logical extension of the Computer Aided Software Engineering(CASE) tool where computer software were envisioned to help create computer software. Fuggetta[28] provided a classification of the different CASE tools based on their level of process integration. The three categories of *Tools*, *Workbenches* and *Environments* shows the level of process support a CASE application can provide. Tools alone provide the least level of process support where the bulk of the tool's execution is for supporting specific and technical support of generating code. Workbenches provides a simple integration of several tools where the focus is on allowing them to work together. Environments provide an integration of Tools and Workbenches where their intricate interactions provide better process support. Progressing from Tools to Environments, the point of focus has evolved from technology into the support for processes which are more undefined.

A PSEE provides an environment that allows process models to be enacted. The term enacted is used, rather than execute, in order to differentiate the view that processes should not be view as only executing within a machine but are operating within the informal human domain. Enacting a process can thus be described as an execution of human and machine processes and the intricate interactions between them. The result is a PSEE which uses software in order to support human processes. The environment can provide the level of process support according to what Madhavji[45] terms as *descriptive*, *prescriptive* or *proscriptive*. Descriptive models involve the description of a current process where the intention is to just model the current state of the process. Prescriptive models on the other hand provide the *defined* process, which is the view of a desired process.

Proscriptive process models operate by prohibiting inappropriate programmer actions. This provides a more tighter integration of the agents within the process. Here the user is made aware of the process by some form of feedback to guide and manage user actions within the process. In this manner, the agent is always aware of their role in the entire process. A PSEE is supposed to provide both prescriptive and proscriptive views.

Sommerville[87] attempted to provide an explanation as to why PSEEs have not been well adopted in real-world usage. He contends that initial attempts have

mainly been too focused on the technical aspects of providing an environment for process control rather than effective process support. The view is that process control assumes that workers within the process conform exactly to a set of rigid procedures. Social processes provides the unknown and 'softness' factor that must be catered for by the environment. This observation goes some way in accounting for the reasons that current approaches, of using a programming language to describe processes, have been relatively ineffective.

These views are echoed by Warboys et al [98, 99] who suggest that process modeling can benefit from the ideas from organisational theory and cybernetics.

1.1.5 Some Current PSEEs

Some environments that are still being used, albeit more in a research environment are Little JIL[90, 91], Apel and Process*Web*[103]. A description of Process*Web* is provided as it is the current PSEE that is accessible for our experiments.

Process*Web*, is a web-based front end for the ProcessWise[16] PSEE. ProcessWise itself was derived from the IPSE2.5[95, 94, 85] project with the purpose of creating an integrated environment for process modeling. ProcessWise can be seen as consisting of three elements, the Process Control Manager(PCM) which supports process enactment, The User Interface(UI) Server which allows the process models in the PCM to provide a UI to users and the Application Server which provides an interface for extending the PCM with external tools.

The PCM forms the core, where models written in a language called Process Modeling Language(PML), can be enacted. PML is an object-oriented language which defines the basic classes for modeling Roles, Actions and Interactions. PML also supports the dynamic compilation of PML where the initial state in the form of variables of the evolved process is maintained and reinserted into the newly compiled process. This facility provides a very powerful approach to supporting process evolution. Process models written in PML are also persistent as long as the process is still referenceable from the persistent root. The property of persistence proved to be quite useful especially in preserving a process even during a hardware failure.

1.1.6 Some Characteristics of PSEE

PSEEs are built to support what Conradi[19] defined as Human Oriented Systems where computer based systems and humans interact to achieve a common goal. This assumes that humans are themselves treated as tools or agents within the system[95]. In order to describe and enact a process, a PML and a PCM is used. As earlier PSEEs were more focussed in providing an environment for executing processes, the initial environments were focussed more on re-adapting the software development tools that were available then. This resulted in, for example, the APPL/A [89] language being based on the ADA language with some added constructs such as Relations, Triggers and Predicates.

PSEEs are also required to integrate a vast amount of tools which developers have been using. Anderson[4] described a set of key open systems and integration mechanisms such as CDIF, PCTE and the ESF software bus.

CDIF supports integration by describing a common data interchange format which uses abstract data schemas to define exchange data but does not specify the structure, content and interpretation by the tools. PCTE supports data integration via a shared data repository and the use of published schema that describes the structure and nature of the data but does not specify the interchange protocol. CDIF and PCTE are thus deemed as complimentary technologies.

The Eureka Software Factory(ESF) provide a client-server approach where tools provides services to each other in response to messages received along what ESF describes as a software bus.

These integration mechanisms are meant to be as generic as possible as PSEEs often need to interoperate with many different classes of external applications.

1.2 Evolution(Softness) in Process Modelling

Sommerville[86] described the need to support informality in the software process but noting that most software process modeling paradigms did not cater for this. He justifies this by arguing that human systems are hard to formalise as they are undefined and are prone to continous changes. He noted that though the increase in formality had contributed to an initial improvement in quality and productivity of software production, as the software artefact gets more complex over time, the benefits from the application of formality has diminished. It is hard to foresee much more improvement by suggesting more formality in the software

process. Sommerville thus suggested that formality should be only used in order to standardise and understand the underlying structure but should not be used to constrain the emergent properties of a process.

1.2.1 Some Characteristics of Evolution

Lehman[40] describes a software system as being of different types. He differentiated them in terms of their potential for evolution. The labels given are the S-type and E-type systems.

S-type systems assumes that operational parameters are completely catered for. This implies that the assumptions that have been made during the creation of the system are static and will not change. This is the dominant view that has been adopted by engineering methods as it is often easier to construct systems where all the operational parameters are known.

E-type system, on the other hand are systems which are continually evolving. The intuition is that the current methods of analysing, specifying and constructing software results in systems that solve an outdated problem as the problem domain, in which the software is supposed to function, has already evolved. Most process models tend to be of an E-type system as they operate predominantly within a human environment.

In a later publication, Lehman[42] revisited and reconsolidated all of these rules. The 'Uncertainty Principle' that is inherent in the design and implementation of software systems that functions in a human domain will always be valid as the assumptions are made in order to create a bounded system. However, the operational domain is unbounded and thus the assumptions made during system design and construction might well have changed.

This view is echoed by Beer[9] who quoted Ashby's Law[6], 'Only change can contain change', in order to deal with the problems posed by complexity and evolution. This idea can be extended to software, where the 'softness' property provided by software is used to manage some parts of itself.

Madhavji[45] proposes the concept of a *process cycle* that embodies both the scope of engineering and evolution of software processes. Separating the process cycle into three sectors that is based on the responsibilities of the process users, Madhavji labels them as A. 'engineering process models', B. 'managing software processes' and C. 'performing software processes' The key aspect of this view is that feedback is generated and received about the process from sector C.

Robertson[76] argued that process engineering should adopt an evolutionary approach as the modeled domain, ie the real-world process, is dynamic and thus cannot be solved by traditional engineering approaches. The justification for this is that the traditional engineering cycle of understanding the problem, then designing the solution to the problem and implementing the solution assumes that requirements are complete, correct and consistent. Moreover, the artefact that has been created as the solution is often non-adaptable. In this sense 'maintenance' could be aptly named 'replacement' of the software as it usually involves a fixing a bug/feature by replacing/removing portions of the software that have been deemed as not useful.

In summary most approaches for supporting evolution require some form of a feedback loop that allows the system to be actively monitored so that the model is as close a fit as possible to the relevant real-world process.

The support of evolution in software artefacts provides a key problem that must be addressed in the software process. Software by its nature is malleable [14] however it has generally been been built using hard engineering methods which were more effective for creating physical and tangible products. There were attempts to make software more concrete by applying mathematics to the problem, but the notation itself is complex and this makes verfication an extremely difficult task. The application of formal notations to software is a good step towards the construction of better software. However, the shortcomings of previous notations is that they view the software artefact as a static entity. Current formal notations such as the π-calculus [53, 54] attempt to rectify this by introducing the concept of the mobility of interactions in order to model the dynamic nature of software.

In view of these problems, the best approach is perhaps to make use of software that allows its softness property to model the software and to manage the common problems normally associated by this softness. This view is not too distant, as already, there are attempts to provide automatic model checking tools. There are currently, however, insufficient studies on how effective and useful they are in the real-world especially if they were to dynamically model check an executing system.

1.2.2 Managing Evolution Complexity

Modelling the evolution process is a complex undertaking even for very simple models. There are however different approaches to manage this complexity. They

are described in this section.

1.2.3 Meta-Process

The meta-process can be defined as a process that manages another process. This definition is purposely reflexive due to the realisation that the process that monitors and controls another process is itself a process. The concept of meta-processes are evident in self-adaptive structures, flexible middleware, feedback control systems and self monitoring systems. The novelty of a meta-process is that it itself is a process which is specialised to monitor and manage the target processes.

Robertson[75] gave a simple example of a simple banking process which itself evolves over the course of its enactment and interaction within the environment in which it is operating. The crucial difference made between the meta-process and that of the target process, which he refers to as the operational process, is that the meta-process is able to install *ad hoc* changes to the operational process where as the operational process could only change in the way that it was originally programmed.

This does not mean that the process being monitored will not be able to affect the changes on its meta-process, but this is by design rather than a definite feature. The scope of this work does not specify this but rather the aim is to discover an enabling technology that should allow further explorations on these issues.

For this investigation, it is useful to define the software process as consisting of both a production process and an associated meta-process. This results in the following formula:-

$$\text{software process} = \text{software production process} + \text{meta-process}$$

1.2.4 Hierarchical Structures

While the previous section explored and argued for the benefits of incorporating support for evolution within the software itself, an issue which was not addressed was the complexity of such an approach. Even without including the support for evolution, the complexity of current software already systems presents a huge problem by itself.

This view is excerbated by viewing the software artefact as merely a construction process. However, a better approach to significantly reduce the complexity of incorporating evolution into a process model is to view the construction process as an evolution process. By this, the approach is that any software process has its associated evolution process. This approach is compatible to Ashby's law which was mentioned before. The approach also has the added benefit that the evolution process is always present. In contrast, in the traditional model the evolution process is only embedded within the human process. This human process is then applied to projects or, in some instances, translated into project rule documentations.

Complexity can also be reduced by structuring both processes, operational and the evolution, in a hierarchy. Whyte[101] suggested that human thought, both conscious and unconscious, must have arisen during the same time as the human mind was able to organized thoughts in a hierarchy.

The use of hierarchical structuring in the field of computer science is well described by Dijkstra[25] in his seminal paper which described a way to structure an operating system in terms of layers. He also introduced the concept of semaphores as a guard to shared resources by multiple processes. Parnas[65, 66] described a set of criteria based on the concept of information hiding for decomposing software into a set of modules where information hiding was desirable. The importance of a hierachical structure are thus crucial to managing complexity and these ideas will be reflected by the consistent structuring over the layers of a csa-based application.

1.3 Formality(Hardness) in Process Modelling

Mathematics has been used as the language of precision for most branches of traditional sciences. The precision and rigour offered by the language of mathematics provides an unambigious format that preserves the semantics of the specification. Properties of the system to be reasoned about are the properties that are exhibited by the specification and can thus be checked. What is more important is that this allows a specification to be machine readable and thus machine checkable.

Mathematics is also a language of abstraction and of thought but this causes some issues as not all thought can be formalised. Even though software is created to run on a machine that should be based on the principles of mathematics, the

domain of its execution is to serve needs whereby a mathematical model could not yet exist and is not directly apparent. Most traditional mathematics, with the constraint of having a rigid and complete model, will thus not be suitable. Thus, Parnas[68] introduced a type of predicate logic where partial functions are allowed where the resulting truth value of such functions is "undefined" in contrast to the traditional predicate calculus where the resultant value is either a true or false.

Further, Broy[15] described an example technique of using a current software development notation with an associated formal notation for each of the non-formal methods that has been used.

Some of the key reasons that formal methods techniques were not widespread are due to their being too verbose, hard to specify, hard to reason about and most usage has been limited to only very simple and low level systems. It might well be that in order to prove that a very large system meets the criteia of safety and liveness, the task to prove this will be longer than the usefulness of the system in the application domain. This is the reason that most formal methods have to date, been limited to only long running mission-critical systems where it is more important to get it right the first time rather than constantly allow it to evolve over the course of its execution.

Formal approaches are however useful especially as process models have the tendency to be complicated and hard to reason about. For this investigation, formality is assumed to be provided by the process modelling language itself.

1.4 Hypothesis

Having provided a summary of the area of research, the main problems faced within the area, some previous attempts at resolving the issues and the potential approaches that will drive the research, a hypothesis for this investigation can now be derived. The research into process programming and PSEEs as a solution for supporting real-world processes are still relevant today. There is, however, a huge gap between the claimed and real value of current PSEEs as effective tools.

This work takes the premise that the reason for the lack of effectiveness of PSEEs in supporting real-world processes is due to the large gap between the real world process and its associated PML model which is enacted by a PSEE. The main issue is due to the rigidity of the enactment system and its representation

system. This is also due to a PSEE not supporting or not being able to provide the correct level of support for the continuous feedback that is driven by the imminent evolution requirement of the process domain.

In short, the view is that most current PSEEs were not designed to be compliant to the application, ie an evolving process model. Even PSEEs, that support some form of evolution, were built to be compliant to the original process domain, and the domain itself is vulnerable to forms of evolution which were not originally envisaged.

A more concrete and detailed definition of compliance will be provided in a later chapter, but for current purposes, compliance can be initially defined as the ability of the support environment to provide sufficient mechanisms to support the policy needs of the process models. Secondly, a PSEE application that is considered as being compliant to the needs of a process model is able to evolve to a form where it will remain compliant to the changing needs of the process model which models the constantly changing real-world process domain.

To provide a summary of the hypothesis, Figure 1.1 shows a diagramatic view of how a compliant systems architecture(csa) is designed to provide better support for the real-world process domain compared to traditional PSEEs. The Physical view highlights that for conventional systems, the only form of interaction with the process domain is focused solely at the process model layer. In contrast, a csa-based system allows a richer form of process domain to process model interaction by allowing this interaction to influence the execution of the underlying software layer.

The resultant logical view highlights that a csa-based PSEE could be better tuned to model the process domain by allowing the customisation of the software at all levels as compared to the traditional PSEE where customisations are only provided by specifying the process model. The intuition is that the traditional approach assumes that the PSEE is immutable.

In summary, this work attempts to verify the following hypothesis, that a system architecture, that has been constructed in the manner where it is defined as being compliant to the needs of the process application to be supported, provides a more flexible environment. Further, that this compliant environment is better able to support process models that have the ability to evolve, even outside their anticipated scope of evolution, as compared to one that is non-compliant or does not have the notion of compliance designed into the system.

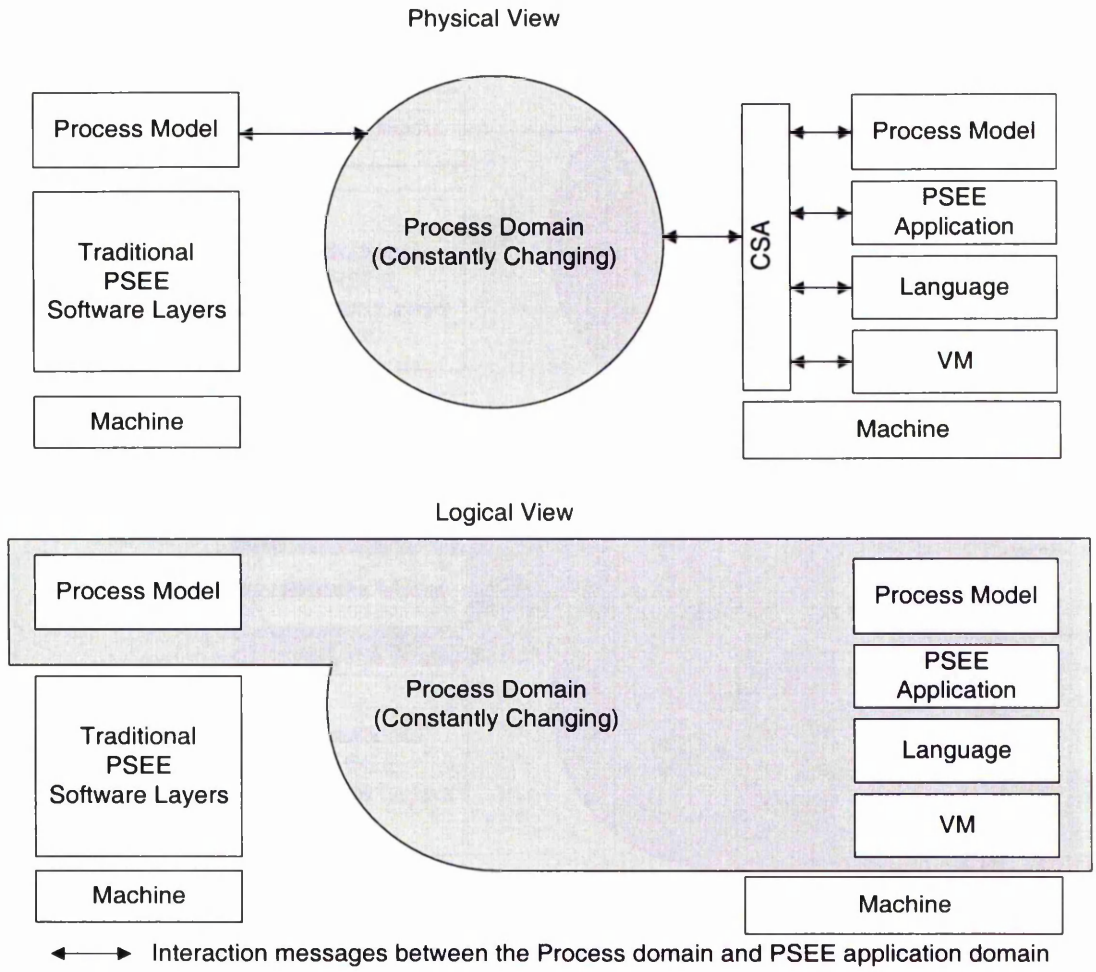

Figure 1.1: Compliance and environment flexibility

1.5 Research Methodology

This chapter started by describing the field of research followed by an exposition of the key problems that still needed to be addressed.

In summary the proposed approach is that of combining the best 'hard' and 'soft' properties offered by software in order to construct a more flexible PSEE for supporting real-world processes. The hard(engineering+formality) approach is still applied for building and structuring a software system in that formality is used as a form of feedback through the checking of properties. This would normally require the use of a formal language together with the use of model checking tools. Formality is utilised in areas to create simple and well understood models which are supported by a meta-process which itself is created using a

formal approach. The softness in this case arises from the dynamic interaction between the created process and its meta-process. In this view, the application is given access to its soft properties instead of it being constrained, a view which is also echoed by Warboys[98].

The research methology is driven by three major phases. The three phases and their associated objectives are described as follows:-

1. Exploration and Definition - The objective of this phase is to provide a definition of compliance and what properties a systems architecture must possess in order that it can be defined as a compliant systems architecture(csa). A more concrete definition of a csa would provide a better understanding of the Compliant Systems Architecture(CSA) approach for designing and constructing systems that are compliant to evolving needs.

2. Construction - The objective of this phase is to test and refine the construction of software layers that are compliant to the needs of a PSEE. The construction of the system layers would also serve as a feasibility test for constructing different compliant system layers which are then integrated to produce a complete compliant systems architecture. The complete csa forms the PSEE which is used to execute process model applications.

3. Evaluation - This phase concerns the evaluation of the CSA approach by integrating the compliant layers. This is used as a test to discover if a system that has been constructed following the CSA approach is better able to support evolution by virtue of its ability to be highly customisable. The construction of a csa itself provides an evalution of the development approach.

To achieve the research objectives for each phase, the following list describes the associated tasks:-

1. Exploration and Definition

   (a) Studying the characteristics of Compliant systems.

   (b) Provision of a more concrete model of compliance that is more suited for describing compliant system architectures for a PSEE.

   (c) Mapping the Compliant Attributes into mechanisms and policies in order to generate a Compliant Systems Architecture.

2. Construction

   (a) Define and construct a compiler in order to study Language Compliance and derive a set of criteria that can be used to test for language compliance

   (b) Define and construct a Virtual Machine in order to study VM Compliance and to derive a set of criteria that can be used to test for VM Compliance.

   (c) Define and construct tools in order to study Application tool Compliance and to derive a set of criteria that will be used to test for Application Compliance. This will involve the construction of a HyperCode system which is customised for the for the study of Language Compliance. The HyperCode System is introduced in chapter 2.

   (d) Construct a Compliant Systems Architecture for a PSEE that utilises each of the already mentioned compliant system layers.

3. Evaluation

   (a) Evaluation from the construction of a csa-based PSEE

   (b) Evaluation from application of different evolution types on the csa-based PSEE

1.6 Research Contributions

It is expected that the results from this work will be directly relevant to the field of software process modeling and enactment. In particular suggesting an approach for constructing a flexible architecture that allows a PSEE to better support all forms of evolution.

The following list details the expected contributions from this project:-

- A more concrete definition of Generic Compliance which is useful for describing and differentiating a system which is compliant from that which is not. This results should include descriptions of the properties that a compliant system must have.

- A set of core attributes for constructing a CSA with a PSEE being the example application.

- An evaluation of a PSEE constructed using the CSA with that of a current PSEE. Presentation of evidence to support that a compliant architecture is better able to support the flexible real-world process which results in deviated process evolution.

1.7 Thesis Structure

The thesis structure mirrors the phases that were described in section 1.5. Chapters 1-3 describes the work undertaken to support Exploration and Definition, Chapters 4-6 describes the Construction theme and Chapters 7-8 details the Evaluation and Conclusions respectively. Each chapter is described in more detail.

Chapter 1: Introduction - The current chapter provided an overview of the field of process modeling and an outline of the core concepts, definitions and some identified key problems. A survey of process technologies and the main problems which affects current PSEEs were also presented. This chapter also described the hypothesis, the research methodology of testing the hypothesis and some expected contributions from this undertaking.

Chapter 2: The Compliant Systems Architecture(CSA) Approach - The CSA[1] approach which will be used for constructing the experimental application will be detailed in this chapter. The approach supports the notion of constructing a compliant systems architecture by realising a software application into mechanisms and policies. A description of the CSA tools that will be adapted for constructing an architecture that is compliant to the application will also be provided in this chapter.

Chapter 3: The π-SPACE Language - This chapter provides a detailed description of the formal Architectural Description Language(ADL) that has been designed for specifying process models. π-SPACE is based on another process algebra, π-calculus [54], with additional constructs for specifying architecture elements. In addition, it also has specific dynamic operators for specifying the evolution of architecture elements.

[1]The "Compliant Systems Architecture CSA Phase 2" project was an EPSRC funded project at Manchester(GR/M88945) and St Andrews(GR/M88938), which was a continuation of the CSA Phase 1 work. More information is available from http://www.cs.man.ac.uk/ipg/csa.html and http://www-ppg.dcs.st-and.ac.uk/Projects/CSA2/

Chapter 4: Language Compliance - This chapter details the work that was required in order to design a language, based on π-SPACE, that can be compiled and enacted. The application of the csa model for determining compliance within the language layer will also be provided in this chapter.

Chapter 5: Virtual Machine(VM) Compliance - The theme of compliance is explored within the context of VM Design and Implementation. The mechanisms and policies and the decisions underpin the choice of mechanisms and policies will be described to justify if the VM can be considered compliant to the needs of the application. The application of the csa model for determining the level of compliance for VM Design and Implementation will be detailed in the chapter.

Chapter 6: Application Tool Compliance - The criteria of compliance within the context of the tool application will be explored here. The application tool provides a process agent with the interface to the underlying PSEE. In essence there are only two basic components that are required, an interface for specifying process models and an interface for specifying the meta-process.

Chapter 7: Evaluation of Compliance - This chapter provides an evaluation of the csa approach for building flexible systems. The criteria for each of the layers will be consolidated to ensure if the architecture has been built to be compliant to the application, the PSEE. The resultant csa-based PSEE application will then be tested against different types of evolution. These evolution scenarios are designed to investigate if a csa-based PSEE is better able to support inherent evolution of process models in the PSEE.

Chapter 8: Conclusions - This chapter provides a summary of the problems and sums up some conclusions and understandings that can be derived from the work. Some potential future research avenues are also provided in order to provide a continuity to this work and more importantly to establish the relevance of this work with other areas.

1.8 Summary

The understanding and modeling of processes presents both an opportunity and a caveat for improving the approach of constructing software systems. The opportunity is exemplified by the ability to mold the software through its interaction with its external environment. The caveat is to understand the complex underlying problems which allows the PSEE to support this type of required flexibility which deviates from the pre-set notions of evolution that existing PSEEs were built to support. Current PSEEs simply failed to support this form of evolution which explains the lack of adoption of such environments in real-world software development.

This work proposes an approach to contructing PSEEs and more generally to constructing an environment that is built on a systems architecture which is compliant to the needs of evolvable software systems. This implies that the entire system, which consists of the underlying operating system all the way to the interface is constructed in a manner that is defined as being compliant to the application. This approach demonstrates a method for constructing environments that are able to support deviated evolution, a form of evolution that was not expected and thus designed into the initial system.

The results of this work are documented in the remaining chapters.

Chapter 2

The Compliant Systems Architecture(CSA)

2.1 Introduction

This chapter provides a description of the Compliant Systems Architecture(CSA) approach developed as a concept during two EPSRC projects aptly named CSA1 and CSA2[1]. The novelty in the approach is that it provides a method for constructing flexible systems architecture that are defined as being continuosly compliant to the needs of supported applications.

A definition of *generic compliance*, a term used in [57], is firstly given. This will then be followed by a review of the literature that focuses on the definition and use of compliant structures in other research areas. In particular the use of compliant structures within the fields of Mechanical/Manufacturing Engineering and Robotics will be briefly explored. The purpose of the review is to study the use of compliance and the properties that define the meaning of compliance in other engineering fields. This will assist in a derivation of a more concrete definition of compliance that is suitable within the context of software engineering. This then results in a definition and model which can be used for determining the compliance of constructed software systems.

A summary of the properties that are required for constructing a systems architecture that is compliant to the needs of a PSEE is also provided. As the

[1]The CSA project was a collaboration between the Informatics Process Group, Department of Computer Science at the University of Manchester and the Persistent Systems Group, Department of Computer Science at the University of St Andrews. CSA1 Grants, GR/L32699 and GR/L34433 and CSA2 Grants, GR/M88945 and GR/M88938

CSA toolset was used to construct the test prototype in this experiment, the chapter concludes by giving a description of the CSA toolset with particular emphasis on their ability for supporting the construction of a csa.

2.2 The Compliant Systems Architecture Approach

Current software applications are built by firstly building the underlying foundation and then layering the required functionality required by a selected generic set of supported applications. This bottom-up approach is mainly due to initial approaches being more focused on discovering the feasibility of constructing a specific structure and on how to construct a core set of logical layers based on the machine layer. The idea is that in order to construct a generic architecture, a concrete architecture has to be built as a prototype that can be used to explore and understand the core issues. This understanding can then be used as a basis for constructing a generic architecture. This approach would generate a valid static layer if the domain and scope of the application is well-known and defined. However, this is the exception rather than a rule for human systems. This approach often produces highly generic layers which might not be suitable for all classes of applications. The assumption that the generic layers are static might not hold true for all classes of application especially in the case of a PSEE as the domain within which it operates demands a more flexible underlying core.

In view of this, the CSA approach takes a top-down approach to focus on the needs of the application in order to construct a core set of underlying layers. This approach is analogous to that proposed by the Requirements Engineering process, that is to create a set of requirements which should be met by the system. Requirements are however abstract and worded in an informal format which are themselves open to interpretations and hence the top-down approach needs some built-in flexibility.

A general view of the CSA is shown in figure 2.1. A compliant system is represented as a set of layers with are composed of mechanisms and policies. Within a particular layer, for example layer *i-1*, policies are bound to a set of mechanisms. The set of policies that are bound will themselves form the mechanisms for the corresponding layer above. For our example, this means that the policies at layer *i-1* will be the mechanisms for policies at layer *i*. This model of compliance is

applied consistently across all the layers within the system.

In this manner, the CSA approach proposes a simple model which is sufficiently flexible and yet concrete to model a system at all layers.

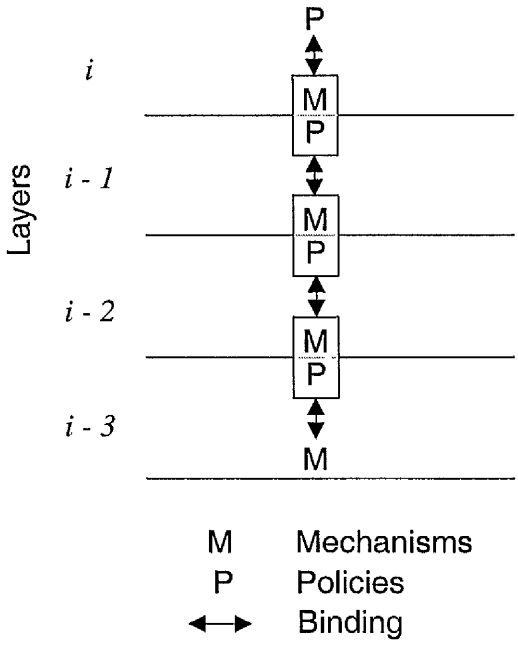

Figure 2.1: The CSA model Layers of Policies Mechanisms and Binding

This is coupled with facilities for reconfiguring the layers so that the property of compliance can be maintained. A definition is provided in the next section in order to clarify the definition of a compliant systems architecture(csa). Of particular interest are the inherent properties that allows a csa to treat the generic layer as a dynamic system that is still prone to change.

A clarification of the difference of csa and CSA is required at this point. csa(*small caps*) refers to a particular instance of a systems architecture that has been constructed to be compliant to the needs of an application. CSA defines the approach, the project and the tools associated with the EPSRC project which developed the initial concept, ideas and tools.

2.2.1 Generic Compliance

Morrison et al[57] described a csa as one that provides the 'best-fit' infrastructure for the intended application. To meet the needs of an application, a csa is layered by separating the mechanisms and policies of components in the applications. These basic components form the basic elements needed to view a system at all levels.

Policies are defined as goals or objectives to be achieved. An example of a policy need for a PSEE is the support for process evolution.

Mechanisms are defined as the means by which policies can be achieved. Example mechanisms for supporting the evolution support policy are the specific meta-process for detecting the need for evolution, the repository for storing the evolving models and a reflective compiler for installing and enacting the resulting model.

Policies can be bound to a set of mechanisms by a *binding rule.* The binding rule is described in the form of a set of *upcalls* and *downcalls* between the policy and one or more mechanisms. The binding rule is thus bi-directional which is compatible with the model as proposed for active systems, that is feedback is an essential property of an active system.

The novelty of the CSA approach is that the mechanism information in *upcalls* and policy information in *downcalls* does not need to be encoded in the same language. Each layer in the systems architecture can be implemented using their native calling convention utilising the language that the layer supports. For example, an operating system written in the C [36] language, might implement the downcall to the underlying mechanism as a C function call and the upcall from the mechanism as an interrupt call to be handled by the policy.

A system with generic compliance will thus has a system layer n with the following property:-

$$policy_n \oplus_n mechanism_{n-1} = mechanism_n$$

where:-

- $policy_n$ is the policy for layer n
- $mechanism_{n-1}$ is the mechanism for layer n-1
- $\oplus_n$ operator is the binding rule for $policy_n$ and $mechanism_{n-1}$

- *mechanism* $_n$ is the result from the application of the binding rule

An interesting result of this property is that a policy at layer n that has been bound to an underlying mechanism at layer *n-1* will result in the mechanism for layer n.

To determine if a layer is compliant to the needs of the policies above it, the application of the binding rule must result in a set of mechanisms that meet the policy needs at the layer above. A binding rule that results in a set of mechanisms that does not provide for the policy needs is deemed to be non-compliant.

This provides an approach for constructing a csa that has been derived from the needs of the application. The actual process of implementation can still be performed following a bottom-up manner but in order for the system to be compliant, the implementation process must be driven by the top-down needs of the supported application.

To further clarify this point, in order to maintain system compliance, each layer must be compliant to the layers above it. The $\oplus$ operator defines how the policy and mechanism are bound together to provide another mechanism for the upper layers. This is compatible with the concept of maintaining some 'transparency' to the underlying functions of the virtual machine that was described by Parnas[69] some decades ago.

A compliant system is thus defined where for all layers, the property for compliance at each layer is true.

Morrison[58] listed an approach for deriving the layers that can be used to construct a compliant systems architecture. The approach requires the following to be specified:-

1. the number of layers in the architecture
2. the system functions required, eg, recovery, scheduling, clock ticks, etc
3. the method used for specifying policy information
4. the method used for passing information between layers and system functions(up-calls, down-calls, horizontal calls)

The definition provided for Generic Compliance is however too generic and abstract a model for determining and constructing a csa-based system. However, the definitions provided for Generic Compliance do provide a good point of reference in order to explore its novelty further.

2.2.2 Definition of Compliance in Other Research Areas

Two research areas where the definition of compliance can provide some insight into deriving a more concrete definition of compliance are from the fields of Mechanical/Material Engineering and Robotics. Both research areas advocate the need for compliant mechanisms and approaches which result in end-products that are better suited for the intended application and thus are easier to adapt to the changes within their individual domains.

Mechanical/Materials Engineering

Ananthasuresh[3] noted that compliant materials have less parts and better strength and durability when compared to non-compliant materials that were designed using conventional engineering approaches. Compliant materials are described as having the following characteristics in contrast to non-compliant materials:-

1. Flexible and strong rather than rigid and strong
2. Are more reliable as they possess fewer moving parts
3. More resilience yet flexible enough within the domain in which they operate
4. Integrated so well with the surrounding domain in within which it operates

Ananthasuresh used the term 'compliant mechanism' in order to describe an *adaptive* structure that was designed to operate as part of the domain in which it is operating. It achieves this flexibility by utilising structural deformation to transmit force or deliver motion due to an input. This is in contrast to being that of a *rigid* structure that consists of actuators, sensors and feedback controls in order to simulate a system with the facility for compliant geometrical adaptation for different conditions. Rigid systems are thus cumbersome, energy inefficient, expensive and unreliable due to the range of different components that need to function in order to simulate the functions that are available in a compliant structure. The underlying structure of a compliant structure is designed to be inherently adaptive(or compliant) which results in a final design that will require fewer actuators and provide a more precise control over force and motion.

The two types of compliance that are relevant here were described as *distributed* and *lumped*. The term *distributed compliance* was used to describe materials where their flexibility is distributed equally across the entire mechanism. In

this manner no one portion of the material is thinner than the other. In contrast, *lumped compliance* uses rigid link mechanisms to simulate the flexibility exhibited by distributed compliance.

The interesting summary from looking at the research conducted into compliant materials suggests that they are 'grown', constructed and adapted from materials that are compliant to the environment within which the material is operating. This is as opposed to the traditional approach of first constructing a set of objects that are foreign to the environment and then combining these sets of foreign objects to form a finished product that has been force fitted to the environment.

An example illustration where the use of compliant mechanisms is not only useful but crucial can be found in the materials that are used to manufacture the wing of an aircraft. The shape of the wing directly affects the lift and performance of flight of an aircraft that is operating in ever changing conditions. A wing built using compliant mechanisms are better able to adapt to the required structural changes in order to continue operating effectively within changing operating conditions. In contrast, an aircraft wing built using conventional methods would utilitise joints, seams and hinges in order to provide the required structural changes. Not only would the wing not be able to respond effectively to changes within the operating environment, the use of joints, seams and hinges would result in discontinuities over its surface. These discontinuities result in a wing with undesirable fluid dynamics which not only reduces the effective performance of the wing but would render it unsuitable for flight.

Robotics

In the field of robotics, the term compliance has been used to describe how the parts that make up a robot fit together such that they interact with the environment in a compliant manner. Most of the problems faced within the field of robotics are quite similiar to those faced by the construction software. Robots are dynamic entities which are constructed using engineering approaches and they are normally used to operate within a human environment. Some examples of the type of compliance will now described.

Mason [46] described how compliant motion of manipulators can be produced either by *passive mechanical compliance* built into the manipulators or *active compliance*(also known as *force control*) which is implemented in the control servo

loop.

The force control is based on:-

- manipulator which is described as the ideal effector and
- task geometry which is described as the ideal surface

A formal model of how the control units map from the language to manipulate these variables was also given. Mason describes that "compliant motion" is achieved by programing a robot to react in a graceful manner when it comes into contact with other objects.

Kosuge et al[37] proposed the concept of *structured compliance* that can be used to describe the compliance of planar parts mating between two parts. The compliance is defined as a model over a generalised coordinate system which also takes into account the friction and positioning errors between the parts. The concept of structured compliance is defined as applying the compliance by applying it over a specific coordinate system.

A good example of compliance in robotics is its used for building a robotics arm. Generally, compliance in a robotics arm refers to the displacement of the wrist in response to the force that is applied on it. High compliance means that a robotics arm is displaced more with a relatively amount of force. The use of compliance in a robotics arm is thus an attempt to model the complex range of movements of a human hand and the dynamic readjustments that the hand would need to make when it makes contact with another object.

A Refined Definition of Generic Compliance

The definition and use of *compliance* within two other engineering fields clearly provides a more concrete foundation for deriving a refined definition of compliance that is useful in the area of software engineering.

Both fields have the key advantage of being well developed and researched with concrete mathematical models that closely reflect their corresponding physical objects. This is to be expected as physical phenomena is easier to observe and thus measure than the phenomena that is generated indirectly from the execution of abstract software artefacts. However, a few insights can still be derived from this survey that can help to refine the model of compliance.

Compliance is described as being a best-fit or well integrated solution where the product performs as a natural extension to its domain of operation. Even

though both fields describe the need for compliance in their materials, it is apparent that an element that is compliant must be related to something else. This could be within the domain in which it is operating or related to other elements which are of the same or different type. The terms used were *structural compliance*, *distributed compliance* and *passive compliance*. The CSA definition of *Generic Compliance* is similiar to these definitions of compliance as the focus is on creating a structure that meets the needs of the application within the scope of its operating domain. Clearly, one could argue that this definition of compliance is no different than the approach taken in creating modular software. The distinction lies in the explicit separation of mechanism and policy across the different software layers, and the requirement that the result from the application of the binding rule will result in a mechanism that will **remain** compliant to the needs of the policy of the application.

Another term that describes a property that is useful for our definition of compliance is that of *active compliance*. Structural compliance by itself allows only changes that has been pre-determined and catered for by the structure during design. Active compliance allows for more adaptability to changes that are external to the original operating domain.

This is catered for by the inclusion of a dynamic feedback control loop in order to dynamically maintain structural compliance even when the domain of operation has changed. This type of compliance seems to be unique to systems that can detect and support reflective changes. It is thus not surprising that active compliance is prevalent in robotic systems and consequently required in highly flexible systems.

Lehman et al [41] contends that there are major differences of feedback within software process systems from those of classical feedback control systems. A model of the Process Feedback Control Model(taken from [41]) is shown in figure 2.2.

The key difference in software development processes from the fields where classical feedback control systems has successfully been applied, is that a large proportion of software development processes are design rather than production processes. Design processes involve a more informal and creative component which demands a more elaborate model than the simple feedback control loop. From the diagram, this means that a PSEE is more prone to changes as the range of I is more. The result is that the O and R will be more elaborate which

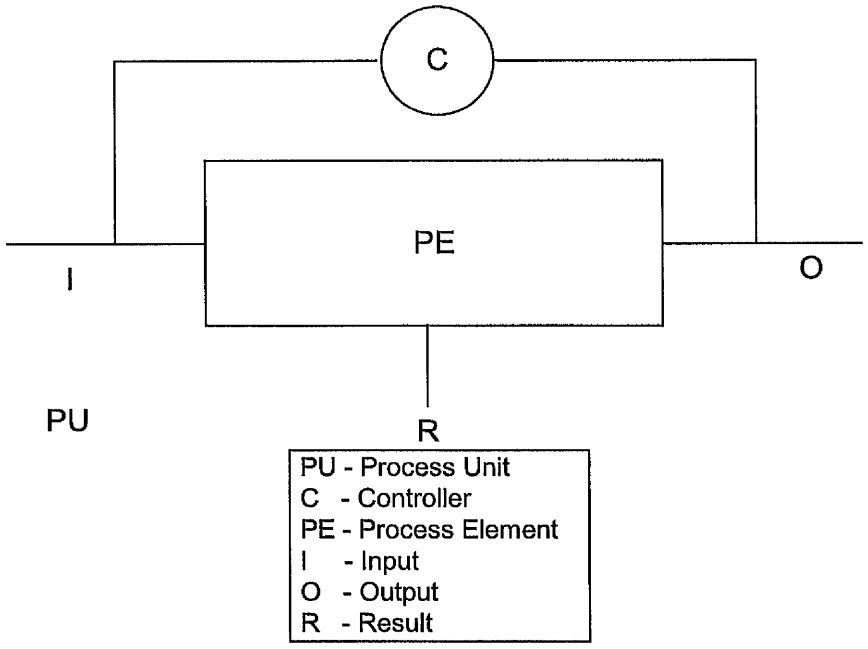

Figure 2.2: Process Feedback Control Model

translates to the requirement that the C will itself have to be more elaborate to handle this increase in complexity.

The conventional model of control theory is based on the explicit premise that change can be planned. Dalcher [22] however, described that complex systems, in particular software systems which are by their nature soft[98], malleable [14] and tractable rarely repeat themselves and thus more concerted effort is required to decipher and plan the inherent dynamics of the interacting systems and their effects on the overall system. The dynamic relationship between feedback, planning and control that is proposed by Dalcher, suggests that in order to support the changes in the system, each component within this interaction should be amenable and thus customisable to change. Instead of incorporating the spectrum of changes that the system can support, the entire system itself should be open to change. This principle underpins the theology of the CSA approach.

2.3 Determining Compliance

A definition of compliance can now be provided which incorporates the original definition of *Generic Compliance* from the CSA project and the use of compliance from different fields of research. A formal model is now described to provide a

more concrete definition of compliance which can be used to determine if a system can be termed as being compliant to some property.

The first rule of a compliant system is that, it must be described in the same form as that of the problem domain. Within the context of a software application domain, this assumes that the application must be described in the form of mechanism, policy and binding rule. The Universe of Discourse of a such system can be given as follows:-

Definition 1: A system is a 3-tuple(M,P,B) where

1. M = Set of all mechanims
2. P = Set of all policies
3. $\oplus$ = Set of all Binding Rules that maps the p to m where p$\in$P and m$\in$M

The binding rule must consist of a downcall and an upcall where policy information can be passed downwards and mechanism information can be passed upwards respectively. The policy and mechanism information need not be the same form or type for all levels. The only constraint is that policy and mechanism information must be compatible and useable by the immediate layer where policy and mechanism information are exchanged. Each of the tuples within a compliant system is further described in the following definitions.

Definition 2: A binding rule $\oplus$ is a 2-tuple (u, d) where

1. d = downcall, a function that maps p to m, d(p) $\rightarrow$ m
2. u = upcall, a function that maps an m to p, ie u(m) $\rightarrow$ p

where m $\in$ M and p $\in$ P

The measure of compliance is defined by the mechanisms providing sufficient support for all the already defined policies. The definition of providing sufficient support is determined by the existent of a binding rule that maps each policy to at least one mechanism. If a binding rule does not exist for a particular policy, this implies that there is no mechanism that is able to support the policy and thus the mechanisms available are not compliant to the policy needs.

Thus, the definition of a Compliant Systems can be defined as:-

Definition 3: A system (s) is defined as being compliant(Γ) to the needs of the policies when:-

{$\forall$p $\in$ P: $\exists \oplus_n \in \oplus$: n=number of layers in the system}

where $\oplus$ = Binding Rule which maps all P to M

In order to simplify the measure of compliance, a function can be introduced with the following definition:-

The Determination of Compliance function Γ requires three parameters, M, P and $\oplus$ and returns the boolean values of True and False depending if the parameters satisfy the definition of a Compliant Systems as provided in **Definition 3**. Thus the determination of compliance function Γ can be defined formally as:- Γ:{M,P,$\oplus$} $\rightarrow$ T,F where
Γ(M,P,$\oplus$) $\rightarrow$ T $\Longleftrightarrow$ condition of compliance detailed in Definition 3 is achieved.

In this way, the determination of system compliance can be defined as a binary function which returns true when the above condition for compliance is met and false otherwise.

The measure of compliance can be extended further in order to provide a more fine-grained measure of a compliant system. Two different types of compliance are directly apparent.

As most systems are conceptually realised in the form of layers, the concept of layer compliance is useful. The property of *layer compliance* defines that a layer of software support must itself be constructed from a set of compliant mechanims. Layer compliance is thus defined as:-

Definition 4: Layer compliance is a specific type of compliance where the measure of compliance is applied to a layer of the system. The set of mechanisms, policies and binding rule of a compliant layer are represented as M_l, P_l and $\oplus_l$ respectively where l is defined as a particular layer within a system.

A Compliant Layer(L_c) is a software layer within a system whereby the property layer compliance holds true. The property layer compliance is defined by the rule which states that there exists a binding rule for each available mechanism within the layer. The property of Layer Compliance is described formally as:-

$$\forall p_l \in P_l : \exists \oplus_l \in \oplus, \forall m_l \in M_l: \Gamma(m_l, \oplus_l p_l) \rightarrow T$$

In order for a system to be defined to have achieved *system compliance*, the system must be constructed from layers of software where each has defined to have achieved layer compliance. The measure of system compliance can then be applied on the individual layers by checking the existence of a binding rule for all the policies of the upper layer to that of the mechanisms at the lower layer.

Definition 5: A compliant systems architecture can be defined as a system that is composed of one or more compliant system layers where each layer l is

defined as a compliant system architecture when the following condition holds true:-

$$\forall l, l \in L: \Gamma\ (\ M_{l-1}, P_l, \oplus_l\ ) \rightarrow T$$

All previous measures of compliance are required in order to achieve *active/dynamic compliance* which utilises all the measures of compliance in order to dynamically monitor and determine if the current system remains compliant to the needs of the real world application. This form of compliance is achieved by the use of a meta-process that allows the system to continually be changed in order to reflect the changes in the real-world system. The model for this meta-process is influenced by a feedback control loop which is similiar to the control loop as described within control systems theory.

An illustration of the final model of a compliant system that consists of the main components that must be available in a compliant systems and the binding rule which provides the interactions between the components is shown in diagram 2.3

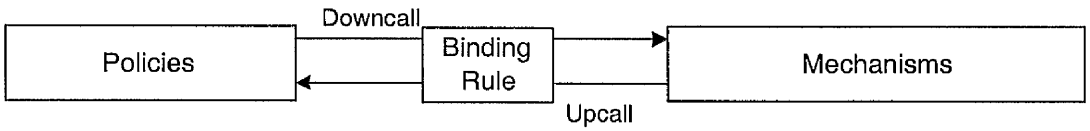

Figure 2.3: A model of the csa showing the required components of a compliant system

2.3.1 Additional Properties of Compliance

Clearly, this definition of compliance exposes some conditions when a systems architecture might be defined as non-compliant to the application. Some other forms of non-compliance arises when policies and mechanism are duplicated replicated across the system. The duplication of mechanisms means that there are more than one instance of the same mechanism implemented within the system. Duplication could either occur within the same layer or at other layers.

In order to specify that a non-compliant system does not have duplicated mechanisms, some further constraint on the model of determination of compliance is required. In addition to the definition of a compliant system in Definition 5, the following rule must hold if there is to be non-duplication of mechanism within a system:-

$\forall l : m_l \in M$ is unique

2.4 A Definition of Compliance for PSEEs

Having described compliance in general, the policies that determine the compliance that are relevant for constructing a PSEE are now described.

2.4.1 Policy requirements of a PSEE

The main purpose of a PSEE is to support the development and enactment of processes. As such, two basic policies are required for any PSEE in order to provide process support. The basic policies of *Process Enactment* and *Communication* are required.

The Process Enactment policy describes the basic primitives and abstractions of the processes supported. Some examples of these are roles, tasks and actors. The policy also specifies how the processes are executed, for example how processes are scheduled according to the execution model.

The Communication policy describes the types of interactions between the process elements that are available. Communication policies involve, for example, the communication protocol and some communication properties such as blocking or non-blocking transfers.

Both these policies are well defined and thus well supported by the underlying mechanisms of current PSEEs. However, in order to ensure that the PSEE is kept up to date to the needs of the process model where its requirements are driven by the human processes domain, a third policy that of *Evolution Support* is required. This defines the meta-process that is responsible for monitoring the feedback from the operational process model, deciding on the response on the feedback and the reinstallation of the required changes on the operational process.

The csa-based PSEE will need to be constructed from many compliant layers where each layer should be compliant to the needs of the top-most three policies of *Process Enactment*, *Communication* and *Evolution Support*. Each layer will have their own policies that are derived from these three policies.

The property of *passive compliance* or *structural compliance* may be sufficient for supporting applications that are either static or only support changes where they can be pre-determined or anticipated. Process models however have the

inherent ability to evolve in ways that cannot be easily pre-determined. To support such evolution, *active compliance* is required. This property requires that a compliant structure must itself provide a reflective component which allows itself to be reconfigured to maintain compliance with the evolved application. This suggests the requirement for a meta-process which is similiar to a control system that has a complete feedback/install cycle to ensure compliance.

The notion of *active compliance*, a term often used in the field of cybernetics, should provide a suitable approach for supporting the evolution of process models which require changes that were not pre-determined during the design of the PSEE. It should also be noted that the policy needs are not an exhaustive list and in fact they are vulnerable to changes themselves. However, the evolution policy might provide a sufficiently flexible approach to cater for these changes.

2.5 Description of CSA Tools

The CSA project produced a set of tools that can be adapted and used for constructing compliant architectures although these were never really exercised by applications within the timescale of both CSA projects. The toolset was designed to provide a basic system architecture that can be customised to meet the compliant needs of an application. In order to achieve this purpose, each CSA tool was designed and constructed to support the construction of a csa. This set of tools is termed *compliant tools* as the tools themselves could each be viewed as a key component of a csa. Compliant Tools support the construction of a csa by providing the basic underlying mechanisms to support structural compliance. Within the CSA project, the compliant tools consist of:-

- A compliant Operating System, ArenaOS.
- A compliant PML with a compliant abstract machine called ProcessBase and PBAM respectively.
- A compliant systems development environment called the HyperCode System.

2.5.1 ArenaOS

The Arena[48, 47] operating system(OS) is built by utilising a very simple nanokernel that can be configured by Hardware Objects(HWO). A HWO provides a generic interface to access hardware resources. Device drivers interact with the HWO. Policy information is specified in the form of Resource Managers(RM). Resource Managers interact directly with the HWO in order to achieve their goals. Example RMs are the Networking RM, Scheduler RM, I/O RM and etc. From the viewpoint of CSA, the ArenaOS provides RMs for specifying policy information and HWO for defining the mechanisms.

2.5.2 ProcessBase and PBAM

ProcessBase[56] is a strongly typed language with orthogonal persistence. It was designed as the simplest class of programming language for constructing process support environments. ProcessBase can be extended at two levels. The first is by writing new libraries in ProcessBase. This is similiar to the manner in which the C language can be extended using libraries written in C. The second method provides more access to the underlying abstract machine. This method allows the introduction of new opcodes or new parameters to existing opcodes to the underlying abstract machine. Essentially this allows the abstract machine to be reconfigured to support the policies needs that were not originally available in the core language.

Whereas the downcall is provided by function calls written in ProcessBase, the upcall to the language is provided by the use of exceptions and interrupts in the language. Exceptions are run-time errors during the execution of ProcessBase and interrupts are messages from the OS. Both types of upcalls can be detected within the language which allows a ProcessBase another level of flexibility to deal with any form of feedback which is available at the lower software layers.

The ProcessBase Abstract Machine(PBAM)[55] is the abstract machine that has been defined to provide the enactment portion for models written in the ProcessBase language. A virtual machine has been constructed in the form of a ProcessBase interpreter that translates and executes the PBAM opcodes on the native machine.

2.5.3 The HyperCode System

The purpose of the HyperCode [105, 106] approach is to unify the representation and entity domains of a software system. This is achieved by providing a single representation of both domains and by introducing a set of operations in order to keep both domains consistent with each other. The domains are classified as the representation(R) and the entity(E) domains. The R domain denotes the source representation of the system whereas the E domain represents the corresponding domain of entities that contains all the first class values defined by the programming language together with various denotable non-first class entities such as types, classes and executable code. Entities in the E domain can be classified as executable and non-executable. Consequently, the R domain can be partitioned into a set of representations of executable entities and a set of representations of non-executable entities.

In order to make sure that the state of the same entities in the R and E domains are consistent at any time, Vangelis[105] also defined a set of domain operations that are required for defining a basic HyperCode System. These operations are detailed below:-

- reflect - reflects the model from the R domain into the E domain
- reify - reifies the current state of the entities within the E into the R domain
- execute - executes the model in the E domain
- transform - allows transformation of the R domain

Figure 2.4, taken from [105], shows the conceptual model that describes how the domain operations are used to ensure that the E domain is consistent with the R domain.

The HyperCode System is made up of a HyperCode Assistant(HCA) and a HyperCode Server(HCS). The HCA provides the interface for the user to create and interact with HyperCode models. The HCS provides the basic operations which implements the basic HyperCode Operations. The HCA allows the viewing and construction of programs by representing the entities as text and hyperlinks. Textual representation provides the static description and hyperlinks provides a reference link to dynamic entities which can be constantly changing. This mix of text and hyperlinks together form the HyperCode Representation. The HyperCode Representation is manipulated by HyperCode Operations(HCO).

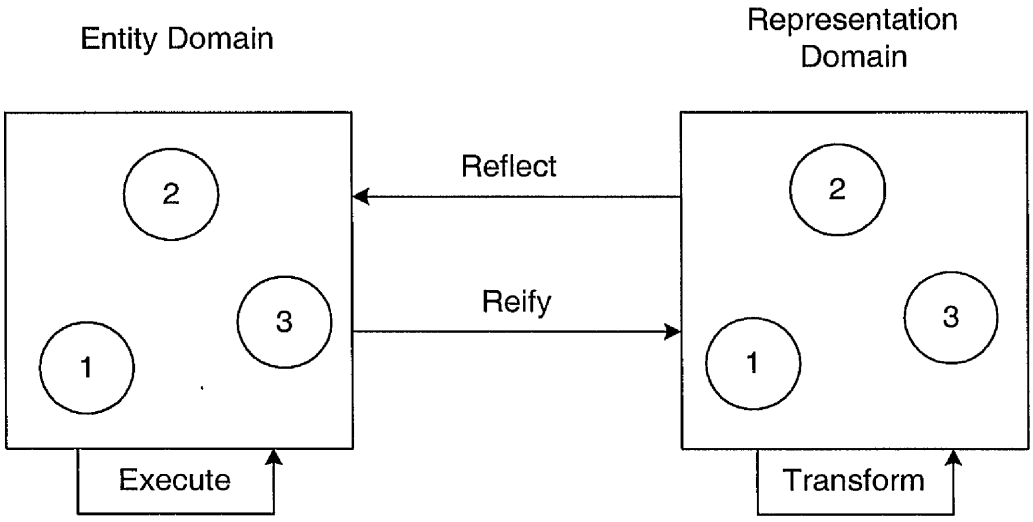

Figure 2.4: Conceptual model of a HyperCode System

There are five basic HCOs which are provided by the ProcessBase HyperCode System. They are listed as follows:-

- Implode - Implode is used to return a hyperlink into its original view after an Explode(see below) operation. This also means that an Implode operation cannot be applied to a hyperlink that has not been exploded.
- Explode - This operation exposes the details within a HyperCode representation. For example, if the hyperlink is a reference to a value, the actual value at the point of 'exploding' the link is returned.
- Evaluate - Allows HyperCode Representations to be compiled into actual executing entities in the Entities domain. The HyperCode representation is compiled and then executed immediately if the compilation is successful. Depending on the result of the execution, a hyperlink to the result is returned.
- Edit - This operation defines the types of code editing facilities that are used to modify the HyperCode representation.
- GetRoot - This operation returns the Persistent root to the HyperCode system.

2.6 Summary

The CSA approach provides an approach to constructing system architectures that best meet the needs of the application. It is expected that a csa is sufficiently flexible to support the type of evolution that has not been well supported in most PSEE. This flexibility is provided by the separation of mechanism and policy across the entire scope of implementation, ie starting from the application layer through to the underyling machine. A definition of Generic Compliance for CSA was detailed in order to provide a rigour to describe systems that are compliant and those that are not.

The use of compliance in two other areas of research were also explored in order to relate the use of compliance to other more concrete fields of research. This study resulted in a refined definition of compliance where the terms *passive/structural compliance* and *active compliance* were derived. This chapter has also provided a definition of the policy needs which are required to be supported by the mechanisms of a csa relevant to PSEEs. The CSA toolset were described as they are used to construct the csa for a prototype PSEE. The results of this construction experiment is described in the later chapters.

In summary the properties that are required in a system in order to achieve *Generic Compliance* are:-

- Separation into mechanisms, policies and binding rules
- Binding rules in the form of upcall/downcalls to map policies to underlying mechanism, mechanism to policies.
- A Layered view to ease the separation

The measure that determines compliance is determined by the existence of a binding rule with bi-directional calls that maps all policies to at least one or more underlying mechanisms.

A definition of *Dynamic/Active Compliance* was also provided which required a system that has *Generic Compliance* to be coupled with a meta-process that monitors and dynamically attempts to reconfigure itself in order that it will continue to be compliant to the needs of a changing application. The CSA project provided the initial informal definition of Compliance and the CSA toolset. The derivation of the formal model as reported in this chapter and the further prototype development through the use of the CSA toolset which is detailed in later

chapters are work completed to support the hypothesis described in chapter 1.

Chapter 3

The π-SPACE Language

3.1 Introduction

Process models are described by a notation that is normally called a process modeling language(pml). This also means that a pml is the mechanim by which a PSEE is customised to enact a specific process model. A pml is thus a vital link that enables any form of process enactment on any useful PSEE. In this section, the π-SPACE ADL is described. π-SPACE is a component-based Architecture Description Language(ADL) that is based around the π-calculus[53, 54] with additional language constructs for specifying components, connectors and their dynamic behaviours.

This chapter starts by describing the key attributes offered by the use of an ADL and the benefits that they provide for constructing dynamic systems. As the π-SPACE ADL is based on the π-calculus, a basic description of the calculus is provided in order to provide an understanding of how it has been incorporated into the π-SPACE ADL. This chapter will also provide some details of the additional core ADL constructs and operations provided by π-SPACE with emphasis on how they provide an architecture-centric approach coupled with the π-calculus process algebra to provide some rigour for specifying and constructing evolving process support applications. The π-SPACE operations for supporting evolution will be described with an illustration of a example evolution that was originally described in [18]

3.2 Overview

Software Archictures [82, 83] is a progression from the work arising from the need to structure software as a hierarchy[25, 65, 66].

Parnas[67] described the approach of developing program families where sets of programs are constituted of programs where it is worthwhile to firstly classify their common properties and then determine their special properties of individual members. The concept of modules arguably formed the basis for components[92] and module interfaces the *connectors*[2]. However, both components and connectors are conceptual constructs where the focus is on the architecture of the system rather than implementation details. Cszyperski[92] describes a component as a unit for independent deployment, a unit of third-party composition and that it has not state. These properties allows a component to be self-contained and a generic unit where the only way to interact and configure the component is via its interface. Connectors describes the interaction relationships among the components.

Shaw and Garlanl[81] discovered that software architectures are often built up by following a certain style of combining the individual components and connectors. Example styles are the pipe-filter and client-server styles.

An Architecture Description Language(ADL)[44] is used to describe a software architecture. In general, this means that the language is used to specify components and the interactions between them. Medvidovic [50] provides a good classification and explanation of how an ADL is different from other languages. He claims that the differences with other languages such as programming languages, Interface Modeling Languages(IML) and Object Oriented(OO)[11] notations lies in an ADL's focus on a conceptual architecture and explicitly treating connectors as first class entities.

3.3 π-SPACE

π-SPACE[1] additions to the π-calculus are in the form of introducing ADL constructs and wrapping up the specifications to specify the properties of those constructs. As such the benefits of the π-calculus specification are retained and used within the context of architectural components.

3.3.1 The π-calculus as used in π-SPACE

π-calculus[53, 54] provides a simple notation with powerful semantics that provides the rigour that is useful for constructing process based applications. The basic concept is that of a *name*. Names reference values which could either be processes or channels which allow the interaction between two different processes. Milner argues that the π-calculus was designed to facilitate the modeling of systems with concurrent processes and the interactions between them. The concept of *mobility* allows channels to be treated as entities that can be sent along other channels in the π-calculus.

There are other properties such as bisimulation, abstractions, concretions described by Milner. However, these properties will not be dealt in this thesis as they are not relevant within the scope of its use in the π-SPACE that is used. This section only provides a description of the π-calculus which is defined and used within the context of the π-SPACE language.

Primitives

Every process and channel can be denoted by a *name*. Processes have behaviours which are specified by the π-calculus. Channels supports the interaction between the processes by allowing the sending of messages between between difference processes.

Operators

Operators operate on the primitives.

The following are the π-calculus notations used within the π-SPACE language.

[1]The π-SPACE language was created by the University of Savoie at Annecy. The refinements of the language were done over the course of various collaborations with the Informatics Process Group within the Department of Computer, University of Manchester. This section thus describes a refinement of the language that was presented in [18]

For the following examples, we assume that P, Q and R are names of processes, and a and b are names of channels and x and y are just names which could be of any type that is valid in π-calculus.

- Prefix operator ($\bullet$)
  The prefix operator Process followed by another process.

  $P \bullet Q$ means that process P executes before Q

- Parallel operator($||$)
  This operator allows concurrent processes to be executed

  P . ($Q||R$) means that Q and R can execute at the concurrently independently of each other after the completion of P

- Conditional/Sum operator ($+$)
  The conditional/sum operator describes a sequence of processes where either one can be executing. There is no concept of priority on which one is executing.

  For example: $P \bullet (Q+R)$ means that either R or R can be executing after P completes

- Success operator ($)
  This operator is used to specify the condition when a process has succesfully completed its execution and is in a stopped state.

  For example, $P \bullet \$$ means that after the P completes its execution, the process has completed its execution.

- Renaming operator ($/$)
  The result of this applying this operator is that occurences with the name of x are replaced by y within the scope of the operation.

  For example, x / y means that all of x is renamed to y. Renaming is usually in order to allow names to match. This is usually necessary for the send and receive operations.

- Send operator($\overline{channel}$ <*name*>)
  This is a channel operation which sends an entity to another. The receiving channel that receives the entity sent will have the same name with the send channel.

For example, $\overline{a}$ $<x>$ means that a message with the name x is sent via channel with a name a.

- Receive operator (*channel(name)*)
  This operator allows a channel to receive an entity. The entity is sent via the channel name. As the receive operation is essentially a blocking operation, they are often used as a way to synchronise the execution of concurrent processes where a receive operation is used like a guard or condition to begin its execution.

  For example, $b(x)$ means that a message is received via channel b. The receive channel can only receive an entity from a send operation with with the same operation name. For this example it means that there must be an corresponding $\overline{b}$ $<y>$ send operation.

3.3.2 π-SPACE types

Primitive Structures

π-SPACE types can be described in terms of base types and aggregate types. Base types are the most basic level which borrows some of the its definitions from π-calculus. Base types consists of *Entities*, *Channels* and *Operations*.

Aggregate types are composed of a combination of base types. These includes *Port*, *Behaviour*, *Component*, *Connector* and *Composite*.

Each type is described in detailed with some examples in the following sections.

Process Entities

A process entity is the most basic structure that can be specified. Its type can be can that of any type that is valid within the π-SPACE language. The initial π-SPACE which was essentially used for specify properties of systems did not provide a set of types. Every construct in the π-SPACE language is just considered as an entity. Entities are thus only differentiated by their names. This concept of names was borrowed from the π-calculus.

Channels

Channels are the most basic entity that allows processes to send messages with other processes. Channels in π-SPACE are similiar to those that have been defined within the π-calculus.

Operations

Operations provide a way to specify a computation. Their behaviour is specified using the π-calculus. The inputs to the operations are defined by the parameters that are passed to the operation. Parameters has associated access specifiers which are defined as **IN**[*type*], **OUT**[*type*] and **INOUT**[*type*] for parameters that are mean as inputs, output and both respectively.

An example of an Operation type is defined in the example below.

```
define Operation Type abc(in[m1:Module], out[m2:Module])
{
  abc[m1,m2] = abc[m2,m1] + $
}
```

The operation type *abc* accepts takes two parameters where parameter *m1* is only for input and *m2* is the output. This example operation type takes in a variable *m1* and returns it via the *m2* parameter.

3.3.3 Aggregates

Aggregate types are built from a combination of primitive types, *Process Entities*, *Channels* and *Operations*. These are the component-based type abstractions that provides an ADL element to the language.

Port

A *Port* type is an aggregate over a set of *Channel* types. It also contains an associated π-calculus element for specifying the behavioural constraints of its port.

A *Port* is specified in two parts. The first part specifies the name and type of channels from which the *Port* is composed. The type of channels are variants of the base types available in π-SPACE. The square brackets are used to specify

the channel type. For example a channel that accepts a Module type is specified as [Module].

The second part provides a specification in π-calculus over the behaviour of that the set of channels. Effectively, the π-calculus specifies the allowable behavioural constraints of the *Port* which are to be adhered to during component composition.

The following illustration should make this clearer by defining an example Port type.

```
define Port type Request[data1: [Module], data2: [Module],
                                          par1: Module, par2:Module]
{
    Request[data1, data2, par1,par2] = data1<par1>.data2(par2) + $
}
```

Port type *Request* is defined as accepting channels *data1* and *data2* and Modules *par1* and *par2*. The behaviour constraint on Request is that it will send Module *par* over *data1* before sending Module *par2* over channel *data2*.

Behaviour

A *Behaviour* type is an aggregate over a set of *Port* types and *Operations*. There are two types of Behaviours based on their use for Components or Connectors. Behaviours that are used for describing the behaviours of Connectors do not have Operation declarations within them.

The format of declaration for Behaviour follows that of the Port type aggregate but only Port types are accepted.

```
define Behaviour Type Check[supply_check:Request[receive:[Module],
    send:[Module], module: Module, reply:Module ]
{
  Check[supply_check] = supply_check@receive(module) •
                        do_check[module,reply]•
                        supply_check@send<reply>.Check[supply_check]+$,
  do_check[In[module], Out[reply]]{...}
}
```

Behaviour type *Check* accepts *supply_check* which is of *Port* type of with the name of *Request*. The element names of *Request* are exposed clearly in π-SPACE

in order to do name matching. The behavioural constraint as specified by the π-calculus defines that a *module* is received on the *receive* channel and then checked using the *do_check* operation. The result of the *do_check* in the form of a *reply* is then sent back via the *send* channel.

The *do_check* operation is specified within the Behaviour type but it is not defined within the example as it contains the same structure as described in the Operation Type declaration as defined before.

Component

A *Component* type is an aggregate over a set of *Port* types and *Behaviour* types. The Component type also consists of a π-calculus definition that specifies the constraint.

Components thus provides an abstraction over Behaviours and Ports. An example definition of a component type is:-

```
 define component type Check1[supply_check:Reply[receive:[Module],
send:[Module], module: module, reply: Module]]
{
    port supply_check: Reply[receive,send, module, reply] ||
    behaviour check: Check[supply_check]
}
```

Component *Check1* is made up of a one port and a behaviour. Port *supply_check* is bound to the behaviour *check* in this component.

Connector

A *Connector* type is an aggregate over a set of *Port* types and *Behaviour* types. The declaration for a connector type is similiar to that for a component with the only difference being that their definition cannot refer to or contain an *Operations* declaration within them. The reasoning behind this is due to the fact that Connectors do not have any behaviours. If there are any behaviours, they are supposed to be routed to another Component.

Composite

Composite types are aggregates over a set of *Components* and *Connector* types. The idea of a composite allows sets of pre-defined components and connectors to be specified.

3.3.4 Operations on Channels

The operations on Channels are derived from the operations of π-calculus as they have similiar semantics. There are however minor syntactic differences. The types of operations on channels are the *send* and *receive*, and *attach*, *reattach* operations and the *renaming* operations. They are detailed with some examples of their syntax. In the examples, *channel_a* and *channel_b* are names with channel types and *msg* is the name of an entity which is used as a parameter.

1. Send(⟨*channel*⟩), Receive((*receive*))

   - send
     The send has the following form. $\overline{channel_a}$<msg> The result of this specification is that an entity abc is sent via *Channel* channel_ a
   - receive
     channel_*b*(abc) Receives an entity of abc via *Channel* channel_ b

2. *detach, attach, reattach* operator
   These operations provides the equivalent property of mobility in π-calculus. Bound channels are 'moved' across different channels by *detach*ing, *attach*ing and *reattach*ing to different channels.

   The attach operation has the following format:-

   attach *channel_a* to *channel_b*

   The result of this attachment is that channel_a and channel_b will be attached which means that a send on channel_a will result in an entity being sent to channel_b and vice versa.

3. renaming operator(/)
   Renaming of channels works like the same with renaming any entities. It has the following format.
   channel_a / channel_b

where the occurences of channels with the name of *channel_a* are replaced by *channel_b*

All occurences of channel_ a are renamed to channel_b and thus any references to channel_b are actually refering to what was defined as channel_a

3.3.5 Operations on the *Aggregates* type

The operations that operate on *Aggregates* are provided in order to compose a set of *components* and *connections* together and to define their behaviours to support evolution.

The *compose* operation

This operation composes components and connector together to form a composite. The result of a composition is a component which is self-contained if all the ports in the connectors and components are attached. If there are any ports with channels that are not attached, they will form the channels for the resultant component.

The specification of a Compose operation consists of two parts. The first part gives the type of components and connectors that are to be supported. The second portion, which is specified after the `where` keyword, allows the specification of how the components and connectors are composed together. The specification are used to specify the way the ports within the components are attached together.

The following example of a compose operation uses some new names and types in addition. They should be self-explanatory.

```
 compose aComponent
{
  aComponentA:ComponentTypeA ||
  aConnectorB:ConnectorTypeB[caller[receive/send, send/receive],
                             callee[send/receive,receive/send]
  aComponentC: ComponentTypeC

where
  attach aComponentA@request_check to aConnectorB@caller
  attach aComponentB@supply_check to aConnectorB@callee
}
```

A composite *aComponent* is composed of two components and a connector. *aConnectorB* has is made up of two ports *caller* and *callee*. Each of the *caller* and *callee* ports has two channels each, called *receive* and *send*. The name of *receive* channel is renamed to *send* and the name of the *send* channel is renamed to *receive* on the *caller* port. This is done so that the names will match when the request_check port is attached to the *caller* port of *aConnectorB*.

The second portion of the definition which is provided after the `where` keyword specifies that *aComponentA*'s *request_check* port is attached to *aConnector*'s *caller* port.

3.4 Support for Dynamic Evolution

π-SPACE provides additional constructs for supporting the dynamic evolution at the component level. Evolution can either be specified directly, ie meaning that the evolution is just going to happen or can be specified based on certain conditions, ie event based depending on some conditions being achieved. The first involved direct evolution whereby there are no alternatives where as the second provides a more grey area whereby the evolution might not be according to plan depending on the different inputs.

The *decompose*, *recompose* and *replace* operation

π-SPACE constructs can decomposed into their original form before composition. This compositional approach provides a very powerful paradigm that allows components to be built and managed in an incremental manner. The decompose operation has to be specified within a compose operation and the operation can only be applied to a component that is a composite.

Reusing the previous example as illustrated for the compose operation, the following example shows how the original component is recomposed into a new component. This example introduces a new component and connector aptly named *aNewConnector* and *aNewConnector* respectively.

```
compose aNewComponent
{
  decompose aComponent ||
    C3: ComponentTypeD
```

```
where
   replace aComponentA by C3
   recompose (C3, aConnector, aComponentC)
}
```

A new component, *C3*, and type, *ComponentTypeD* is introduced during the composition of component *aNewComponent*. *aNewComponent* is constructed from *aComponent* but with a new component C3 which is of type *ComponentTypeD*.

The dynamic (π) operator and, *where* and *whenever*, constraints

The dynamic operator allows a dynamic number of processes to be specified. For example if a certain component A is supposed to have an indeterminate number of instances, it will have a name $A\pi$. Constraints, conditions that will invoke other operations when the conditions specified are met. These constraints are specified within the *compose* and *decompose* operations. Taking the previous example into consideration, we will now specify a dynamic connector *C2* and dynamic component *C3*.

```
compose aNewComponent
{
  decompose aComponent ||
    C2π: ConnectorTypeB ||
    C3π: ComponentTypeD
where
   replace aComponentA by C3π
   recompose (C3, aConnector, aComponentC)
whenever
      new C3π ⇒ new C2π
      new C2 π ⇒ attach C3π@supply_check to C2π@callee
}
```

The behaviour is similiar to that as described in the previous example except that whenever a new *C3* is created, a corresponding new *C2* connector is created and bound to the new C3.

3.5 Summary

This chapter described the π-SPACE ADL by firstly describing the relevant structures that were derived from the π-calculus. The π-SPACE additions that provide the abstractions required of an ADL language are then described in terms of the entities and operations available. π-SPACE's constructs that were designed to support evolution by way of applying the composition and decomposition operation on constructed components.

However, it must be noted that the π-SPACE language described in this chapter does not assume an enactment component. There has been various attempts to create equivalent models by hand in a simulation language, PICT and a Process Modelling Language, PML [18] but there was no π-SPACE abstract machine that supports the enactment of π-SPACE models. This task will be undertaken as one of the investigation to create a csa-based PSEE which allows some enactment of π-SPACE models.

Some refinements to π-SPACE in order to prepare the specification language for enactment, ie compilable and then for enaction, has been made. This work is described in chapter 4 and its complete BNF and Code Generation rules are documents in appendix A.

Chapter 4

Language Compliance

4.1 Introduction

The definition of Generic Compliance that was provided in chapter 2 applies to the system as a whole. A system is often composed of different layers where each layer can then subsequently realised to be composed of a set of interacting components. In the case of a csa-based system, this set of interacting components are defined in terms of the mechanisms, policies and the binding rules between them. These basic components of a csa-based system were described in chapter 2.

This chapter explores the use of compliance within the language layer where process models are specified using a particular PML that was derived from the π-SPACE language[18]. π-SPACE was defined as a specification language based on the π-calculus by LLP/CESALP Lab, ESIA, University of Savoie at Annecy. A more detailed description of specification π-SPACE was provided in chapter 3. A description of the refinements that were made in this research in order to allow the π-SPACE to be compiled is firstly described. After which, a distinction is made between the original π-SPACE language that is described in chapter 3, hereby known as *Specification π-SPACE*,and *Enactable π-SPACE*, which essentially is a language derived from *Specification π-SPACE* with further refinements to make it machine compilable and enactable. The π-SPACE language was selected as the PML due to several reasons. They are listed here:-

1. π-SPACE is an formal ADL which is based on the π-calculus
   The formality provided by the ADL provides the 'hard' engineering approach.

2. Support for evolution through the introduction of dynamic operations for composing and decomposing systems
   This feature is itself compatible with the CSA view to achieve compliance where a system can be evolved by composition and decomposition of components which together make up the system.

A definition of language compliance will also be given that will be used as the basis to measure the compliance of the resultant Enactable π-SPACE language.

At this juncture, a clarification of the relationships between the languages is required. The original π-SPACE, now termed as Specification π-SPACE, is an ADL which was designed for specifying highly evolvable system architectures. Enactable π-SPACE is derived from Specification π-SPACE in order to refine it into a language which is compilable and thus enactable. Being enactable, the Enactable π-SPACE language thus forms the Process Modeling Language(PML) which is used for writing enactable process models. The Enactable π-SPACE language is compiled into a base language called ProcessBase. ProcessBase is compiled into PBAM opcodes that are executed with the ProcessBase interpreter.

4.2 Design of the enactable π-SPACE Language

The design of the enactable π-SPACE language was mainly driven by refinements made to the original specification π-SPACE language to yield an equivalent enactable format that is suitable for compilation. As this enactable language is designed around the recursive-descent compiling approach[23, 1], a short description of the approach is firstly provided. A summary of the refinements that were made to the Specification π-SPACE language in order to construct the compiler for the Enactable π-SPACE will then be given.

4.2.1 Recursive Descent Compiling

A recursive descent parser is one of the simplest to develop as each non-terminal in the grammar can be mapped onto a corresponding recognition routine. It is so named as the parsing of a language is performed in a top down manner by recursively breaking down a sentence and descending into the corresponding recognition routines for each language construct. This also makes it relative easy for implementing type checking routines as type checking can be applied

during the parsing phase. As such the compiler for ProcessBase and the reflective compiler was implemented using such an approach. For this experiment, the decision was made to reuse the core of the compiler so that the compiler can be easily incorporated as a reflective compiler within the PBAM.

The task of writing the parser for a context free language that has been defined in the Extended Backus Naur Format(EBNF) is also sufficiently straightforward where each terminal symbol in the language can be recognised by the lexical analyser and each non-terminal symbol can be recognised by recogniser functions that can be invoked recursively. The abstractions that are provided by the recogniser functions also helps to manage the complexity of constructing a compiler. Meta-symbols of the EBNF such as '|' and '*', each representing conditional and repetition operations on the symbols respectively, can be easily translated into their equivalent constructs in any programming language.

In addition, the focus on recogniser units also means that at each valid step of the syntax parsing process, the recogniser is able to invoke specific semantic and code generation actions based on the type of symbols that were recognised. This however implies that there is no backing up during the parsing process and that each step of the parser must be deterministic as the recogniser function that is invoked will be determined on the current token that is being read. This places a constraint on the language in that the language must be LL(k) compliant where the k is the number of lookahead symbols that will influence the invocation of which recogniser unit.

In contrast to other parsing strategies, another key property of a recursive descent parser is that an explicit syntax tree or parse trees is not generated during the parse phase. A snapshot of the syntax tree is only implicitly reflected when viewing the invocation trace of the recogniser functions. For recogniser functions that have been written in a language that uses a stack-based storage for storing the activation records of functions, the current structure of the syntax tree can thus be seen in the execution trace of these activitation records.

4.2.2 Lexical Refinements

Specification π-SPACE was designed as a language whose focus was on specifying component properties with sufficient rigour such that the resultant models that were specified in π-SPACE could be checked and reasoned about, if possible by machine-based model checking tools. This specification-biased approach is

directly apparent from the use of mathematical symbols that are usually hard, if not impossible, to represent in a text-based programming notation.

To allow the language to be parsed, some lexical refinements were required in order to make the language easier to specify in a textual format that is supported by a computer. This meant that all the text, including special symbols are assumed to be limited to the ASCII standard.

The following list shows the lexical refinements that have been made to the Specification π-SPACE language:-

- Reserved words
  Reserved words are tokens in the language which cannot be used as identifiers or names. Some reserved words in enactable π-SPACE are as follows:-

  `define component connector operation compose where whenever`

- Constants/Literals
  Specification π-SPACE did not specifically have any pre-defined constants or literals. As such the assumption was made that the constants/literals will be implemented based on the constants/literals that are supported by the base language on which the π-SPACE is to be implemented. In this experiment, the base language was ProcessBase.

- Special Characters
  All special characters where there is a corresponding representation in ASCII text were retained. An example are the angled brackets, <>, that were used for the send operations and the round brackets, (), that represents the receive operation in π-SPACE.

- Identifiers - names
  As everything is basically defined as having a name in π-SPACE, enactable π-SPACE names are composed of characters that are of alphabet type and the underscore character. The names must also not be the same as any of the already defined reserved words.

4.2.3 Syntactic Refinements

Syntactic refinements are changes that were made to the syntax of the specification π-SPACE language so that a compiler could be constructed. There are two

factors that directly affected the design of the syntax for the enactable π-SPACE language.

The first factor is due the approach of constructing the compiler. The decision to construct a recursive descent parser meant that that the language had to be made to be at least LL(1).

The second aim was to adhere as closely as possible to the specification π-SPACE language. This was so that the enactable π-SPACE language could utilise planned model checkers and reasoning tools with only minimal syntactic changes. These tools were expected to be available from other research laboratories that were briefly mentioned in the paper on π-SPACE[18]

In order to achieve both the above-mentioned aims, the following approach was followed:-

1. Simplify the language so that it is possible to construct the compiler within the limited time available for the experiment on constructing a csa-based PSEE

2. Introduce additional language constructs that add enactable elements to the basic specification π-SPACE language

Following this approach, some specific syntactic refinements were made to the specification π-SPACE language which resulted in the design of the enactable π-SPACE language. Some of the more interesting refinements are listed below:-

1. *Removals*

   (a) 'extends' keyword from all the type definition
   'extends' allows a type definition to inherit the properties of its parent type. The decision to discard the 'extends' keyword in the grammar was mainly motivated by the need to simplify the virtual machine design for the language. Inheriting the properties of previously defined objects might be a good facility to have in the future, but for the current experiment, this facility is not important for constructing a csa-based PSEE.

   (b) composite type
   The composite type is composed of components and connectors. The original intent was that when a composite type is instantiated, a whole

set of components and connectors will be created and linked following the specification within the composite. However, the result of this instantiation is still a component as π-SPACE is a compositional language. The only justification for retaining the composite type is to enable the language to identify which components are themselves composed of other components and connectors. This however can be implemented by decomposing the component itself and then testing for the existence of subcomponents.

2. *Additions*

   (a) Annotations

   Annotations were added to the language in order to provide the specification of enactable elements that were not present in specification π-SPACE. Annotations are so-named as they are introduced as a meta-language that does not require any major structural changes to the specification π-SPACE language. Annotations introduced some important facilities which provide the basic foundation of the enactable π-SPACE language. Some key facilities are:-

   - values/literals

     Specification π-SPACE provides the notion of names to refer to any basic components within the language without any syntax which allows a name to be assigned a certain value. In order for values to be assigned to identifiers, the following syntax is introduced

     `let` *identifier* `<-` *value*

   - type instantiation to the language

     Specification π-SPACE provided language constructs for defining the basic types that are available in the language. However, there are no explicit constructs that allowed types to be instantiated. Type instantiation was thus introduced and has the form of

     typename`(` *parameter*`, ...` `)`

     where a *parameter* is made up of a pair of parameter name and value with the following format

     parametername`<-`*value*

     Each instantiation will result in the instance of that type which

can be assinged to an identifier.

- expressions
  Expressions for the base types of ProcessBase were introduced. This includes string operations for the creation, concatenation and comparisons of string types and arithmetic operations for the addition(+), subtraction(-), division(/) and multiplication(*) of integer types.

In order to ensure that the Annotations function more as a meta-language that utilises the definitions in π-SPACE, some guards were introduced so that Annotations will be parsed differently. One key benefit from this approach is that the changes to the π-SPACE are kept to a minimum and that the work to parse out the annotations is thus simpler. Annotations are specified within guard symbols that start with <%ps and ends with %ps>.

Table 4.1 shows an illustration of a component definition and its associated Annotation where the component is instantiated as a value and bound to a name.

(b) Local variable declarations within type definitions
Specification π-SPACE did not require local variables to be explicitly defined within any type definitions, as the names of the local variables are either implicit in the type definition parameter header or within the definition. These variables are required for enactable π-SPACE to allow the compiler to deal with scoping issues. Local variable declarations were introduced into the Port and Behaviour types.

Table 4.2 shows an example definition of the Specification π-SPACE.

Explicit local variable declarations were introduced to facilitate a simpler compiler design. Furthermore, explicit local variables also improve the clarity of a specification hence these changes were subsequently incorporated into the Specification π-SPACE language in Annecy.

3. *Miscellaneous refinements*
Even though the following list of refinements did not result in major changes to the syntax, they did have some influence on design decisions about the compiler.

(a) *Communication Channels - send<> and receive()*

Port and behaviour definitions in π-SPACE

```
define port type Request[request:[string], reply:[string]]
{
  Request[request, reply] = request<service>•
      reply(result)•
       Request[request, reply]+$
};
```

```
define behaviour component type ClientWork[request:[string],
   reply:[string]]
{
  sevice:string, result:string,
  internalCompute[in[Service:string]]{println(Service)},

  ClientWork[clientport]=internalCompute[service]•
    clientport@request<service>•
    client@reply(result)• ClientWork[clientport]+$
};
```

Annotations for instantiating port and behaviour values

```
<%ps
  let ClientPort <- Request(request <-[""], reply <- [""]);
  let ClientBehaviour <- ClientWork(c <- ClientPort);
  let Client_Instance <- Client(p<-ClientPort,b<-ClientBehaviour);
%ps>
```

Table 4.1: Annotations in enactable π-SPACE

Specification π-SPACE

```
define behaviour component type ClientWork[
clientport: Request[request:[string], reply:[string] ]
]
{
  ClientWork[clientport] = internalCompute[service]•
      clientport@request<service>•clientport@reply(result)•
      ClientWork[clientport] + $
}
```

Enactable π-SPACE

```
define behaviour component type ClientWork[
clientport: Request[request:[string], reply:[string] ]
]
{
service: string, result:string, ! local variables decl
internalCompute[in[Service:string]]{println(Service)}, !local op decl

   ClientWork[clientport] = internalCompute[service]•
      clientport@request<service>•clientport@reply(result)•
      ClientWork[clientport] + $
}
```

Table 4.2: Local variables in Enactable π-SPACE

Channels in Specification π-SPACE
$\overline{channel}$< parameter>
Channels in Enactable π-SPACE
channel< parameter >

Table 4.3: Differences of the textual representation of communication channel operations between specification and enactable π-SPACE

The overhead bars over the channel names in specification π-SPACE will be hard if not impossible to type using an ASCII-based text editor. It is also apparent that they are not required to determine if a send or receive operation is specified for a channel. The angled, <>, and round, (), brackets characters that appears right after the channel names are sufficient to indicate if the channel operation over the channel name is sending or receiving a message. To be able to parse this syntax though required the parser needs to be LL(2) compliant as the channel name and the brackets need to be parsed to determine the type of operation.

Table 4.3 shows how a send operation is specified for a channel π-SPACE specification in the original and enactable format.

(b) *Separation of Behaviours types for Components and Connectors*
There were different constraints on the behaviour types that were meant to be used for Component types and those that were meant to be used within Connector types. Behaviours that were meant to be used by Connector types are not allowed to have embedded operations defined within them. The rationale is that connector behaviours should be limited to specifying only processes that operate on communications channels. The simplest solution is to introduce a new behaviour type for each component and connector type. This is achieved by adding the extra keyword 'component' and 'connector' in the behaviour type declaration syntax in order to differentiate the behaviour types.
Table 4.4 illustrates the difference between the original π-SPACE and the refinements.

The result of these refinements made on the language forms resulted in a

Behaviour definitions in Specification π-SPACE

`define behaviour type aBehaviourforComponent[]`
 `...`
 `define behaviour type aBehaviourforConnector[]`
 `...`

Behaviour definitions in Enactableπ-SPACE

`define behaviour` **`component`** `type aBehaviourforComponent[]`
 `...`
 `define behaviour` **`connector`** `type aBehaviourforConnector[]`
 `...`

Table 4.4: Differences of the behaviour definitions for components and connectors between specification and enactable π-SPACE

grammar for the Enactable π-SPACE. The complete grammar in EBNF is given in Appendix A.

4.2.4 Semantic Refinements

The semantics of a language describe the actual behaviour of each language construct that is represented by each grammar rule. The result of the semantics is a set of specifications that can be used to produce equivalent code generation rules that can be implemented on top of a virtual machine. During the time when the language was constructed, the enactable element of the language was still constantly being refined and thus there was no formal notation of the semantics except those derived for code generation. The understanding of the semantics are derived from the enaction characteristics as displayed by the generated code. The full code generation rules are provided in appendix A. Some semantic rules for Enactable π-SPACE are shown and described in narrative form in the following list:-

1. Semantic definitions for each construct of Enactable π-SPACE

   (a) Channel - A channel is a primitive along which messages can be sent. There is no behaviour that can be defined by the channel itself. Send and receive operations operate on a channel but they are specified only in the other Aggregate structures.

   (b) Ports - A port is composed of a set of Channels and the π-calculus

specification describes the pattern of communication behaviour on the channels.

(c) Behaviour - Behaviour types are constructed based on specific Ports types. They also provide optional specification that defines their behaviour in terms of a π-calculus notation that specifies how operations and the channels within the Ports can interact.

(d) Component - The component structure is made up of Ports and Component Behaviours. They can also include locally defined embedded operations.

(e) Connector - A connector is composed of Ports, Connnector Behaviours that operate on the channels of the Ports.

(f) Operations - Operations are like functions that can be written in the base language. Originally π-SPACE specifies the Operations using the π-calculus but this was changed when designing the Enactable π-SPACE. Operations defines simple behaviours that are easier to specify using a programming language. This allows π-SPACE constructs to be a structuring notation and the ProcessBase language to serve to implement enactable units. Operations in π-SPACE thus serve as hooks from the structural portion of π-SPACE to the enactable portion provided by the underlying base language which in this case is that of the ProcessBase language.

2. Type rules

It was decided that type rules for Enactable π-SPACE should only be specified on the type of data that is sent over the channel primitive and on the operations that operate on the channel type. The main reason for this was so that the approach would not be bogged down by having to define a complete type system for π-SPACE since the original π-SPACE did not have any explicity types. The end result was that, during parsing, there are types for each π-SPACE construct defined but they are not checked due to time constrains and also that its low relevance to this project.

- Universe of Discourse
  - Base Types
    (a) Scalar data type of int

(b) Type string is the type of character string
The following type constructors are defined in enactable π-SPACE

(c) For any type T, channel[T], is the type of a channel that contains a value of type T.

– Type grammar

type ::= **void** | **int** | **string** | **channel**[type]

– Type Rule

(a) channel type

$$\frac{\tau \vdash \text{T} \in \text{Type}, \tau \vdash \text{T not} \in \textbf{void}}{\tau \vdash \textbf{channel}[\text{T }] \text{ Type}}$$

(b) channel type construction

$$\frac{\text{t}, \pi \vdash \text{e} : \text{T}}{\tau, \pi \vdash [\text{ e }] : \textbf{channel}[\text{T }]}$$

(c) attach operation

$$\frac{\text{t}, \pi \vdash e_1 : \textbf{channel}[\text{T }], \text{t}, \pi \vdash e_2 \ \textbf{channel}[\text{T}]}{\text{t}, \pi \vdash \textbf{attach } e_1 \textbf{ to } e_2 : \textbf{void}}$$

3. Semantic definitions for Annotations

(a) Instantiation operations
`abc(` *parameter*$_1$, *parameter*$_2$,...,*parameter*$_n - 1$, *parameter*$_n$)
where abc is the name of the π-SPACE component type that has been defined previously and *parameters* are names of instantiated π-SPACE components that are composed within the abc π-SPACE component type.

(b) Variable declarations
`let abc <- 1`
where abc is of type integer and assigned a value of 1.

(c) Basic operations
These operations are directly derived from ProcessBase and have been included as a matter of convenience so that basic arithmentic and string operations can be used instead of having to define them as π-SPACE operations.

- multiplication $*$
- division /
- string concatenation $+$

4.2.5 Code Generation

Code generation rules define the code that is generated for each syntactically and semantically correct language construct. Each code generation rule provides an implementation biased definition in the target computer language that must preserve the semantics that have been defined for each language construct.

In general, the code generation rule for each enactable π-SPACE construct consists of a *Type Definition* and its corresponding *Instance Generator*. The structure is described as follows:-

- A Type Definition in π-SPACE is defined as a ProcessBase view type.
  The general structure consists of the following fields with its corresponding type:-
  1. name:string - The name of the type is retained in this field.
  2. typeid:int - This is an internal typeid that retains the type information.
  3. list of parameters that are relevant to the type π-SPACE type -
- Instance Generator.
  This is basically a ProcessBase function that accepts parameters that are relevant to the associated π-SPACE type and returns an instance of the Type Definition. The Instance Generator might also generate and then bind the dynamic elements that are required when the type instance is generated.

The code generation that is specific for each type of language construct will now be described in detailed in the following sections.

A complete list of code generation rules for enactable π-SPACE is available in Appendix A.

Primitives

- Names - Everything in π-SPACE is referenceable by a name. Thus, every entity, be it an instance or type, has a name associated with it. In order to retain this name, every type defintion as defined in the code generation rule has a *name* field of type string.
- Channel types - As they do not have any behaviour, channel types will only result in a simple code generation that has a basic Type Definition and Instance Generator.

Aggregates

The list of aggregate types in π-SPACE are as follows.

- Ports
- Behaviour
- Component
- Connectors

Each aggregate type is represented as a view type in ProcessBase and they have the same format which is now described.

An example code generation rule for an aggregate type, in this case that of a *component* type declaration is shown in the table 4.5. The complete code generation rules for enactable π-SPACE is given in appendix A.

Both the type definition portion and Instance Generator portion of the definition provides a ProcessBase equivalent that is designed to retain all the information required to achieve the semantics defined for the π-SPACE construct.

Type definitions provides the static portion and provide the relevant fields in which to store the data. This data can be in the form of just basic field types in ProcessBase or, if required, a location to a function which allows the attachment of more dynamic components to the π-SPACE construct.

The Instance Generator provides the instantiation of the dynamic component which binds actual instances of constructs in order to generate an instance which can be manipulated and enacted. The code generation rule that is described provides a lazy binding approach where the attributes in the behaviour are only executed and bound during instantiation.

Defining a π-SPACE Component	ProcessBase equivalent

define component type Writer[..]
{
port request_check:Request[...]||
behaviour write:Write[...]
}
	!! Type definition
	type Writer **is view**[
	typeTag : **int**;
	request_check:**Request**;
	write:**Write**;
	start_behaviour:**loc**[**fun**()]]
	!! Instance Generator
	let gen_Writer < − **fun**(...)
	{
	let Writer_start_behaviour < −**fun**()
	{ ...
	}
	view(**typeTag** < − **componentTag**,
	write < − **write**,
	start_behaviour < −
	Writer_start_behaviour)
	}

Table 4.5: An example code generation rule that shows the type definition and Instance generator in ProcessBase of a π-SPACE component

Executing Units - Operations

Primitive executing units in π-SPACE are specified as Operations. A decision was made to allow Operations to be specified in the base language, ProcessBase, because operations are mapped directly to functions and parameters specified in Operations are mapped onto the function parameter in ProcessBase.

Table 4.6 shows how the operation parameters are mapped to their respective equivalent in ProcessBase.

4.2.6 Enaction Issues

The issues discussed here will form the enactment policies that need to be supported by the mechanisms at the VM layer. The actual underlying mechanisms that are needed to be provided by the layer of software that supports the enactment will be described in chapter 5.

Operation Parameter in π-SPACE in access specifier	**ProcessBase equivalent**
`define operation type anOp [in[aParameter:aType]`	`type anOp is view[typeTag:int; aParameter:aType; operation_fun:fun() ]`
Operation Parameter in π-SPACE inout and out access specifier	**ProcessBase equivalent**
`define operation type anOp [out[aParameter:aType], inout[aParaneter2:aType ]`	`type anOp is view[typeTag:int; aParameter:loc[aType]; aParameter2:loc[aType]; operation_fun:fun() ]`

Table 4.6: Code generation rules for Operation parameters

Process enactment

The processes that are described within the enactable π-SPACE language are viewed at two levels. The basic level includes the execution of the π-SPACE constructs which are available in Specification π-SPACE. Another level supports the additional processes that can be specified in the Annotations and Operations.

Some properties which can serve as policy information are:-

- Process thread priority
- Thread Control, Suspend/Resume, etc

Communication

The communication policies are essentially derived from the needs to support the operations on channels in π-SPACE.

The following are some properties which can serve as policy information:-

- Channels support the protocol of single send with multiple receives.
- Invocation of the receive operation on a channel results in a blocked state on the operation until a message is received.
- Able to attach a channel to multiple other channels.

4.3 Language Compliance

The previous sections detailed the actual design decisions that were applied and the results of the design were illustrated as examples. The purpose of describing the actual design of the language was to understand and identify how the model of compliance can be applied.

To be compliant to the needs of an application domain, a language is required to provide a set of underlying mechanisms that meet the policy needs required by the supported application. Language policies are thus determined by its usage in the application domain.

4.3.1 Compliance in π-SPACE

From mapping the basic components in their syntactic form to their corresponding enactable policies, the underlying mechanisms are then represented by the semantic rules. These semantic rules are then further realised as a set of Code Generation Rules which are implemented in the underlying base language. The approach for constructing a compiler for a compliant language is thus similiar to the approach for constructing any computer based language. The essential difference lies in the fact that policies must be supported by the mechanisms provided in the underlying language.

The only thing left to do is to determine if the enactable π-SPACE is compliant is to attempt to map all the policies to the underlying mechanisms that are provided by the language.

Components

A compliant systems must be represented as a set of P, M and $\oplus$. The following lists the set of Policies P, Mechanims M and Binding Rule $\oplus$ that can be realised with justifications.

1. P = The constructs in the π-SPACE ADL
2. M = The ProcessBase constructs that are designed to provide an enactable element for each π-SPACE construct
3. $\oplus$ = The Code Generation Rules which maps each language construct in π-SPACE onto ProcessBase

Binding Rule

The binding rule must be described in terms of its downcall and upcall.

1. Downcall
   The downcall is in the form of the invocation of the compiler. Policy information are described in the form of the π-SPACE specifications and the compiler flags that can be passed on to the compiler.

2. Upcall
   Result of compilation forms the feedback from the compiler. The mechanism information is reported as the types of compiler messages that are reported from the result of compilation.

Determination of Layer Compliance

In order to determine layer compliance, the Compliance function Γ can be used. The needs of each policy which is a π-SPACE construct are met by an equivalent construct in ProcessBase. As each π-SPACE construct has an equivalent grammar rule, we can confirm that all the policies needs are met by the mechanisms that are implemented as ProcessBase.

Thus the determination of compliance, which is made concrete by the construction of a compiler that implements the syntax and semantics as defined earlier and the conceptual model is said to be compliant to the needs of the policies.

Figure 4.1 describes the model of compliance as applied to the π-SPACE language and its associated ProcessBase equivalent.

4.4 Criteria for Language Compliance

In order to determine that the language is compliant to the policy needs of a process model, the language will have to be used by a process model. In order to determine language compliance, the mechanisms provided by a process modelling language must fully support the policy needs of the process models.

The policy needs of a PSEE language and the mechanisms provided by the π-SPACE language are listed as follows:-

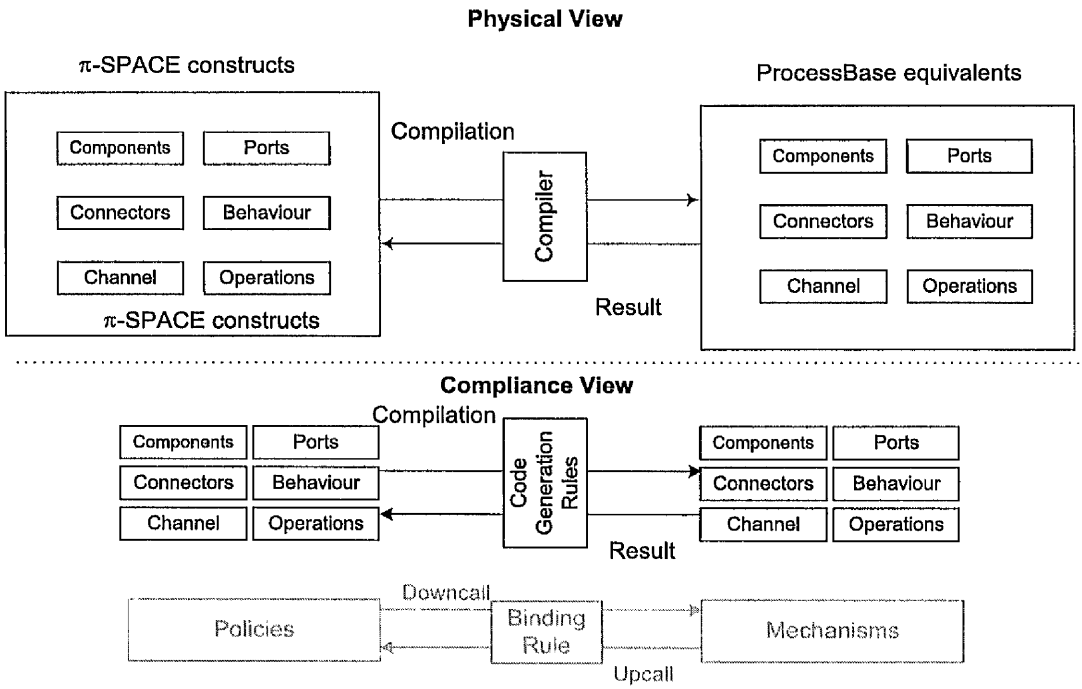

Figure 4.1: Language Compliance, Compiler and Language

1. Process Specification - As a process is the core entity of a PSEE, the ability to specify a process is the most basic policy need required by a PSEE. Process Specification will involve the specification of process behaviour and process interaction.

   The mechanisms provided by the π-SPACE language allows these abstractions to be specified using abstractions such as Component, Ports and Behaviours.

2. Constraint Specification - The behaviours of each process and their interactions can be structured according to constraints. Constraints must be placed on the basic processes in order to provide a structure for the different behaviours that can be exhibited by those processes.

   Mechanisms offered within π-SPACE utilises a form of π-calculus which allows constraints on the constructs provided within the language.

3. Dynamic Evolution Support - This can be considered a special type of Constraint Specification in that it is a type of behaviour that allows the current behaviour of a process to be changed during its enactment.

The π-SPACE language provides support for evolution by introducing the notion of composing and decomposing components and the ability to specify the processes in respond to some evolution.

4.5 Summary

This chapter detailed the work that was required to refine the π-SPACE language into a form that can be compiled and thus enacted on a virtual machine. The definition of language compliance within the context of a PSEE was initially provided. The PML provides core constructs that allows the specification of these policies. The use of π-calculus specification within the PML also provides the facility for specifying the behaviour of the process enactment. Interactions between the processes are provided by the synchronisation facilities of the operations on channel. It should be noted that, this work was not an attempt to redefine a language that will execute π-calculus as this has already been attempted in Pict[71].

A set of policy needs were also derived from the requirements that are required for a PML. These were derived from the PSEE policy needs such as Process Scheduling, Constraint Specification and Communication Handling which were then described. This set of policy needs will be useful for the evaluation of language compliance which is detailed in chapter 7. This evaluation can only be done when the meta layer(above) and the abstract machine layer(below) is constructed to see if the policy needs of the PML is supported by the underlying mechanisms of the abstract machine. To be compliant to the layer above, these policies with the underlying mechanisms must provide sufficient support for that layer.

Chapter 5

Virtual Machine Compliance

5.1 Introduction

In order to execute the resultant ProcessBase code that was generated by the π-SPACE compiler, a virtual machine is required. This chapter begins by providing a basic definition of the term *Virtual Machine*(VM) and describes some key attributes that are available in most conventional VMs. Examples of two real-world virtual machines are then provided as illustrations of contemporary designs. These illustrations are used as a basis for drawing out a summary of VM features in order to highlight the differences of current VM designs to one that is built to be compliant. The design of the csa-based π-SPACE/ProcessBase Virtual Machine(πPVM) is then detailed. The purpose of designing and constructing the πPVM is two-fold. The first is to explore the inherent properties of a virtual machine that is constructed to be compliant to the needs to a PSEE application and secondly to demonstrate how a compliant virtual machine can be constructed using the CSA toolset. Essentially, the πPVM is based on a customised version of the VM for executing PBAM opcodes.

5.2 VM Design Approaches

5.2.1 A Definition of Virtual Machine

Before providing the definition of a Virtual Machine, another term needs to be defined. An *abstract machine* is a processor design which is not meant to be implemented as hardware. Abstract machines are often designed to execute an

intermediate language that has been generated or used in a compiler or interpreter.

Abstract machines can thus be defined in terms of the following:-

1. An instruction set - The instructions are usually designed to be of higher level than the instructions set of a typical processor hardware. The reason is that this will serve as an intermediate language between the underlying hardware and the supported higher level language. The instruction set for an abstract machine is also known as the *Abstract Machine Language*(AML).

2. A set of registers - These are immediate memory locations that can be used directly by the set of instructions. Registers that are defined in an abstract machine need not have a corresponding register implemented in the hardware.

3. A model of the memory - A description of the memory, for example if the memory layout utilises a heap or stack model, that can be accessed by the instruction set.

A *Virtual Machine(VM)* can thus be defined as an abstract machine for which an interpreter is available for executing the language that is supported by the abstract machine. In some instances, the term *abstract machine* has been used interchangeably in place of the term virtual machine.

A VM is defined as a concrete software implementation of an abstract machine that allows the execution of the supported abstract machine code. Popek[73] defined a VM as "an efficient, isolated duplication of the real machine". Using the CSA toolset as an example, this would infer that the PBAM be the abstract machine definition which was described in the PBAM manual[55] and the implemented interpreter, a software designed to decode and execute PBAM opcodes, is the VM.

A key benefit of utilising a VM design is that it presents a consistent interface to the supported application program. This is achieved by abstracting the main features of the hardware such that an application program can be executed on different hardware platforms without the need to be changed and recompiled if a VM has been implemented on a particular platform.

A VM can also implement some features which the hardware or operating system does not currently support. For example, most current hardware does

not support automatic garbage collection that most contemporary computer languages and their VMs such Java[43], Limbo[74], C# and ProcessBase[56, 55] support.

To summarise, the relevant characteristics of VMs are:-

1. They are implemented in software in order to abstract away from the specific hardware
2. They facilitate portability of bytecodes across different hardware platforms
3. The efficiency of the VM to execute code is important as slow execution of object code will negate the the benefits of code portability.

5.2.2 Conventional VMs

The term "conventional" VM is now introduced and defined in order to differentiate it from that of a "compliant VM". Conventional VMs are constructed based on the assumption that they will cater for a selected group of applications. The assumption is valid if the selected group of applications are static. However, application needs are prone to constant change and new applications may also be added to the original set of applications which brings about a need for change on the VM. Most VMs are designed to support changes that are well-defined which can be programmed into the VM.

Two current VMs will be described in order to explore why they are considered as being conventional as they are tuned to cater for a 'generic' set of applications. The term 'generic' however is a misnomer as they are based on the set of applications that are currently known. A more appropriate definition of conventional VMs are that they are tuned for a particular well-known class of applications which makes them inappropriate for classes of applications where their abstractions and needs are in a flux and are constantly evolving. Process models usually belong to the later class of applications.

The Java VM

Java [43] was originally named the Oak project and is a platform designed by Sun Microsystems to run on multiple devices that have small memory footprints. One of the goals of Java was to achieve the goal of 'write once run on many platforms'.

A VM is thus required in order to achieve this goal as writing a different compiler to generate different code for each platform will be an extremely arduous task.

The Java platform is made up of a set of technologies which Sun labels as a Java Software Development Kit(SDK)[52]. There are currently three editions of the Java SDKs, Enterprise, Standard and Micro Editions, where each edition is fine-tuned for different platforms with different levels of sophistication and usage. Each SDK is made up of a set of development tools that includes the Java compiler, profiler and libraries, and the Java Virtual Machine(JVM) which includes the interpreter to execute compiled Java bytecodes.

Extensions to the language are made by adding new features to the core libraries which are usually written in Java. Only major changes to the underlying JVM, for example the addition of a Just in Time(JIT) compiler or other syntactic changes to the Java language, will require a redesign and reimplementation of the JVM.

This model assumes that the available underlying mechanisms are static even though Java does provide support for the upcall in the form of Exceptions[51] in the language. Exceptions are treated as a primitive Class in the Java Language which allows an executing program to handle run-time errors that would normally interrupt the flow of a program.

Exceptions can be viewed as a form of upcall which provides a feedback from the underlying VM to the program. This feedback allows a more dynamic form of customisation which allows different messages to be sent from the underlying mechanisms to the policies implementing as Java programs.

The Dis VM

The Dis VM is designed to run on the Inferno Operating System from Lucent Technologies. Inferno is both a small operating system and execution environment for a wide range of devices and networks that is based on the ideas from the Plan 9[72]operating system.

The Dis VM utilises a Memory to Memory(MTM) architecture instead of a more conventional stack based model. This architecture results in instructions sets that are more natural to current hardware than the instructions sets for stack based machines. However, the tradeoff is that the VM requires a more elaborate interpreter engine in order to parse the instructions than when running on conventional stack-based machines.

The language used for writing programs for Inferno is called Limbo[74]. It has a C-like syntax with some influences from Pascal. Even though it is in theory possible to write Inferno programs in another language that can be converted into Dis bytecodes, Limbo has the advantage of being designed ground-up for Inferno. The use of other languages thus might result in an inability to utilise all the features in Inferno.

A few features in Dis that stands out are:-

1. Automatic Garbage Collection - It uses a more simplified model which provides a good balance between performance and efficiency.

2. Channels are treated as a primitive within the Limbo language

Summary of Contemporary VMs

Having described two contemporary VMs, a summary of similiar key features of contemporary VMs can be derived. This list is used to highlight the different approach that a compliant VM provides in contrast to one that has been built using the conventional approach. This exercise also helps to reveal how a compliant VM can be constructed. The following is a summary of contemporary VMs with the details of how each differs from one that we define as being compliant.

1. The nature of the instruction sets are usually very low level.
   In the case of Java it is understandable as their only goal is to abstract a generic set of processors to ease the porting of the JVM to different platforms.

2. Most VMs are created to support the generic set of applications where their properties are known.
   This assumption means that the abstractions and the features supported by the VM are often tuned and optimised for this set of applications. However, this set of abstractions might not provide the set of mechanisms that will be useful for all domains. For example the type of garbage collection scheme that is embedded within the language is considered as a static entity which is not replaceable or contains no way for the application program to be notified of when the garbage collection is going to execute.

3. Most VMs are created as static monolithic structures with no provisions to support the type of changes that would require more changes to the VM

itself.

This is understandable in current contexts as a VM has to be efficient and most VMs are not meant to be evolvable. Most methods of extending the VM are usually limited to support changes that can be specified within the supported language and are deemed as being 'safe' to be changed without breaking the assumptions that have been made on the VM. An example in Java is that most of the libraries are implemented in Java itself. Java does provide an interface, the Java Native Interface(JNI), that allows programs to bypass the JVM altogether in order to make use of lower level system mechanisms. However, being non-compliant, the underlying mechanisms of the JVM cannot be customised or finetuned. These mechanisms might be reimplemented using JNI and hence the original mechanism might be bypassed and not utilised at all. In the extreme case, one might even use the JVM but actually write another VM feature outside of the JVM using JNI. If this is done, then the abstractions and features provided by the JVM would have been negated.

5.3 Compliance in VM Construction

A prototype was developed to construct a compliant VM for the policy needs defined for a PSEE. This required the use of an abstract machine that was designed to support the notion of compliance. The PBAM is designed to be a highly configurable abstract machine that supports the CSA approach. As the features of PBAM has already been described in section 2.5.2, this section focusses on how the PBAM supports compliance and how being a csa-based VM allows the VM to be highly configurable in order to support the construction of a csa-based PSEE. A description of the specific customisations is provided in order to illustrate the degree of flexibility provided by a compliant VM.

5.3.1 Support for Compliance in the PBAM

The PBAM provides support for the construction of layered systems through the use of libraries that provide different layers of abstractions.

A basic set of system functions are provided by PBAM which forms the default set of core mechanisms available. This however can be extended to provide more

mechanisms if this set of basic mechanisms does not provide sufficient system functions for the intended application.

Describing it in terms of the csa approach, the specification of policy information is provided in the form of data values that can be specified using the entire set of ProcessBase types.

Downcalls are provided as parameterised function invocations to the underlying VM. This is similiar to any function call invocations implemented in other VM platforms.

Upcalls are supported through the interrupt mechanism as supported by the underlying VM. The interrupt mechanism provided by the VM can be customised to be as low-level as required. In fact all interrupts that the VM can handle can be made available. This means that every mechanism that the VM itself can make use of, can also be made available to the policies at the higher layer. The benefit of this is that all interrupts available to the interpreter, for example, interrupts generated by the VM can be caught by the ProcessBase language. Run-time exceptions such as type errors generated on the fly by the interpreter, are also available to the application that makes use of the run-time mechanisms in the VM.

In summary, the PBAM provides the basic support that is necessary to support generic compliance. It can also be argued that most VMs do support this type of generic compliance if we describe it following the approach above. Thus, the ability of a VM to support the construction of a compliant system should probably not be measured in terms of the four basic criteria for generic compliance. The ability of a VM to support a compliant systems approach is determined by the level of customisation supported. In essence the level of customisation supported by a VM denotes the available set of rules for binding policies and mechanisms.

In the case of PBAM, the level of customisation provided by the VM goes further than other real-world VMs as described. In addition to supporting the extension of the VM via libraries, PBAM supports another form of reconfiguration by allowing opcodes to be extended within the VM. This provides another level of reconfiguration where the most primitive underlying mechanisms, the opcodes, of the VM can be fine-tuned if required. This extra level of customisability allows the VM to be more flexible than conventional VMs.

5.3.2 Comparisons of PBAM with conventional VMs

Having described both the characteristics and virtues of conventional VMs and a csa-based VM, distinctions can now be made between them. At face value, in many respects, the VMs do seem to be similiar. The manner in which both types of VMs support the construction of layered software via the use of libraries are the same. Their approaches to supporting extensions to the core language by the use of libraries are similiar. In addition, Exceptions in the Java VM and PBAM both enable support for the upcall from the VM to the application. These similiarities are to be expected as the conceptual underpinings of a csa-based VM are built on top of and complement the basic concepts under which convential VMs are built.

The complementary concept provided by a csa-based VM is the level of customisability. This key difference lies in the ability to customise and extend the opcodes in PBAM. This allows all available underlying mechanisms in PBAM and that of other compliant underlying layers to be exposed to the upper layers. This key attribute assumes that the VM itself is susceptible to change and that all mechanisms that are available in the original VM are susceptible to those changes. Taking a leaf out of Parnas' view of transparency[69] of features available in a VM, a csa-based VM thus provides complete transparency to all its available mechanisms.

5.4 Design of πPVM

The π-SPACE Virtual Machine(πPVM) is a VM that is designed to execute π-SPACE constructs. The approach undertaken to construct the πPVM was through the customisation of the PBAM interpreter in order to utilise all the already available underlying mechanisms which are augmented by the introduction of new libraries. These basic mechanisms are in the form of the basic opcodes and the default libraries. If there were policies which were not supported by this default set of mechanisms, initial attempts were to extend the VM by implementing the mechanisms at the library layer. However, if the mechanisms could not be created by writing a new library in the ProcessBase language, the next step undertaken was to customise the opcodes in order to either modify the existing underlying mechanisms or to extend the set of opcodes available. This facility is useful for extending the VM beyond what can be done using the libraries written

in ProcessBase. The design of the πPVM however would not require the use of extending these opcodes but there are scenarios of evolution which should require these facilities. A summary of the resultant customisations are detailed further in the later sections.

5.4.1 Architecture

The underlying architecture of the πPVM can be grouped into the three major components. The grouping of these units are directly influenced by the policy needs of a PSEE that were described in chapter 2. Each unit has a set of mechanisms that have been designed to support their respective policy needs. The policy needs are Process Enactment Support, Communication Support and Evolution Support.

Process Enactment Support provides the mechanisms that support the type representations for the basic components in π-SPACE. Relevant support operations that are required to manage the components were provided. The mechanisms to enact the process were also provided.

Communication Support provides the type definition for the π-SPACE channel type.

Evolution Support provides mechanisms to support process evolution. The type of mechanisms provided are the support for feedback to detect the need to support the evolution and the ability to install a suitable process in response to the evolution.

Figure 5.1. provides an illustration of the resultant conceptual architecture for the πPVM.

The rest of the subsections describes in detail each group of the underlying mechanisms in terms of its implementation in ProcessBase. These mechanisms can be grouped into two types, those that are designed to support passive compliance and those designed to support dynamic compliance.

5.4.2 Mechanisms to Support Passive Compliance

Mechanisms to achieve passive compliance are implemented as functions in ProcessBase libraries. These mechanisms are designed to support the policy needs that have been described in the previous chapter, chapter 4. They can thus be described by grouping them by the policies that they are designed to support.

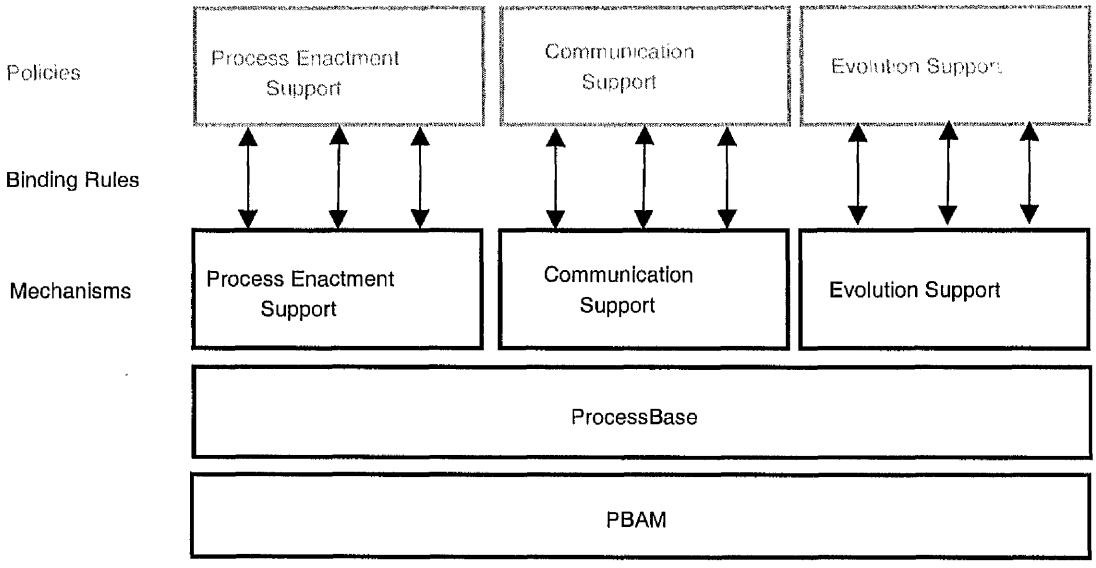

Figure 5.1: The Architecture of the πPVM

Libraries

Most data types have a name field that is of type *loc[string]* which is a location to a string type in the ProcessBase language. The main reason for this decision was due to the requirement that π-calculus, on which π-SPACE is based,treats every primitive as a name.

Any data fields that can be changed over its lifetime are represented as a location(loc) of the type. Types that represent any dynamic structure such as lists or trees are also defined as a location. This usually has the type of the dynamic structure added to the name of the type. For example, *channelList* is a List of Channels.

1. Process Control Support

   These mechanisms provide support for the execution of processes that are specified in π-calculus(see Chapter 3). As there is potentially more than one process executing at one time, they are implemented as threads.

   - Data Types

     ProcessBase Definition

```
ThreadList is view[threadId:loc[int];
    thread:loc[fun()];
    next:loc[ThreadList]]
```

The `ThreadList` structure stores the `threadId` and the location to the function in a list.

- Functions

  (a) Process Creation

  ProcessBase Definition

  ```
  addThreadList <- fun(new_thread:fun();
          thread_list:ThreadList)->ThreadList
  ```

  This function adds a new Thread to the ThreadList.

  (b) Process Instantiation

  ProcessBase Definition

  ```
  let startThreadList<-fun(threadList:ThreadList)
  ```

  This is a function that starts all the threads in the `threadList`. The function makes use of the `ThreadLib` library which utilises POSIX[33] compliant threads.

  (c) Process Removal

  ProcessBase Definition

  ```
  let removeThreadList<-fun(threadId:int;
                  threadList:ThreadList)
  ```

  This function is an inverse of the addThreadList function where a thread is remove from the list of thread based on the id.

2. Communications Control

   Mechanisms are required to support the channel type and the channel operations that are available in π-SPACE. The following mechanisms have been implemented and made available for use:-

   - Data Types

     Figure 5.2 shows the ProcessBase data types implemented and their dependencies for supporting Communication Control in the πPVM.

     – Channel

     ProcessBase Definition

     ```
     view[name:loc[string]; id:int; buffer:loc[ChannelBuf]]
     ```

     This type consists of the name of the channel, a unique `id` and the `channelBuffer` for buffering messages that are received by the

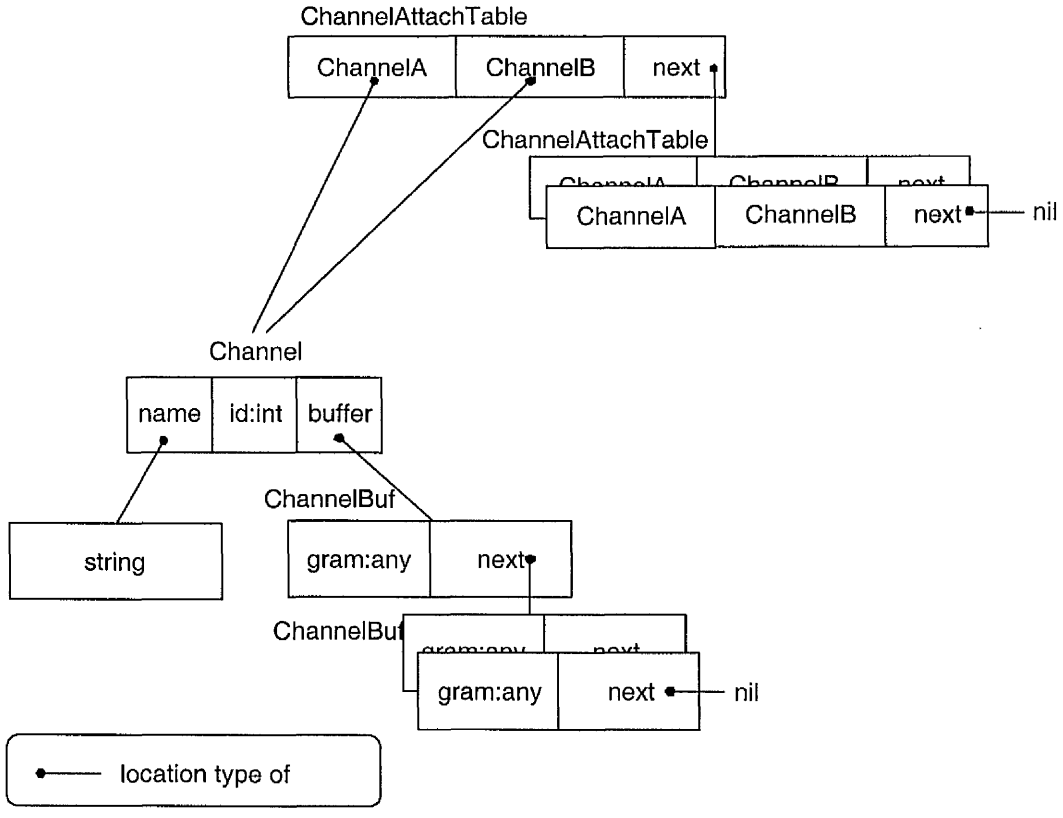

Figure 5.2: Types in Communication Control

channel. The implementation assumes that the mechanisms that support channel communication are implemented in ProcessBase as opposed to using the underlying mechanisms of the underlying mechanisms available in the VM or OS.

- ChannelBuf

  ProcessBase Data Structure

  `view[gram:any; next:loc[ChannelBuf]]`

  This type provides an implementation of a buffering for channels. Each ChannelBuf structure has a gram of type any which allows it to carry a data element of any type in ProcessBase. The next field `next` is due to the implementation decision of implementing the ChannelBuf as a list.

- ChannelAttachTable

ProcessBase Definition

```
view[channel_a: Channel;
     channel_b: Channel;
     next:loc[ChannelAttachTable]]
```

This is a table structure that stores the channels that have been attached to each other. This structure will be used by the functions that handles the send and receive operations.
This lookup table is used to store channels that has been attached. It is implemented as a list.

- Functions
  - Channel Generator -
    ProcessBase Definition

    ```
    genChannel <- fun(name:string)->Channel
    ```

    The function generates a new Channel. It returns a Channel instance with the name specified in the *name* parameter.
  - Channel Attachment via the *attach* operation
    ProcessBase Definition

    ```
    attachChannel <- fun(channel_a: Channel;
                         channel_b: Channel)
    ```

    Attaches channel_a to channel_b by creating a new entry in the global ChannelAttachTable.
  - Sending
    ProcessBase Definition:

    ```
    sendTo(channel:Channel; gram:any)
    ```

    Algorithm(in pseudo-code):

    ```
       Takes channelname as input;
    Scans the ChannelAttachTable(channel_a, channel_b);
    {
      If the channelname matches channel_a then
     add data to ChannelBuf of the associated channel_b
               or vice versa;
    }
    ```

- Receiving

  ProcessBase Definition

```
receiveFrom <- fun(channel:Channel:gram:loc[any])
          -> bool
receiveStringFrom <-fun(channel:Channel:gram:loc[string])
          -> bool
```

  `receiveFrom` implements the π-SPACE *receive(msg)* operation. This is a blocking function which means that if there are no messages in the receiving channel buffer when this function is invoked, the execution thread will be blocked. This allows communication operations to control the synchronisation of threads which is the behaviour that follows operations of the receive operation in π-calculus.

  Algorithm(in pseudo-code):

```
Scan the ChannelAttachTable
If channel matches
{
  If ChannelBuf is not empty, then
  {
      remove gram from ChannelBuf and return as gram;
      return true
   }
  else
      return false
}
else return false
```

- Channel renaming

  ProcessBase Definition

```
renameChannel <- fun(channel:Channel; new_name:string)
```

- Checking ReceiveBuffer status

  ProcessBase Definition

```
checkReceiveChannel <- fun(channel:Channel) -> bool
checkReceiveChannelSize <- fun(channel:Channel) ->int
```

The `checkReceiveChannel` functionchecks if there are any messages on the receive buffer of a channel and the `checkReceiveChannelSize` function returns the number elements currently in the receive buffer of a channel. These functions are required for checking if a channel's receive buffer has received any messages.

3. π-SPACE Structures
   The following lists the π-SPACE types and their corresponding implementation in ProcessBase. The functions available for manipulating these types are also listed.

   - Data Types
     Figure 5.3 shows the overview of ProcessBase types for representing the core types available in π-SPACE.

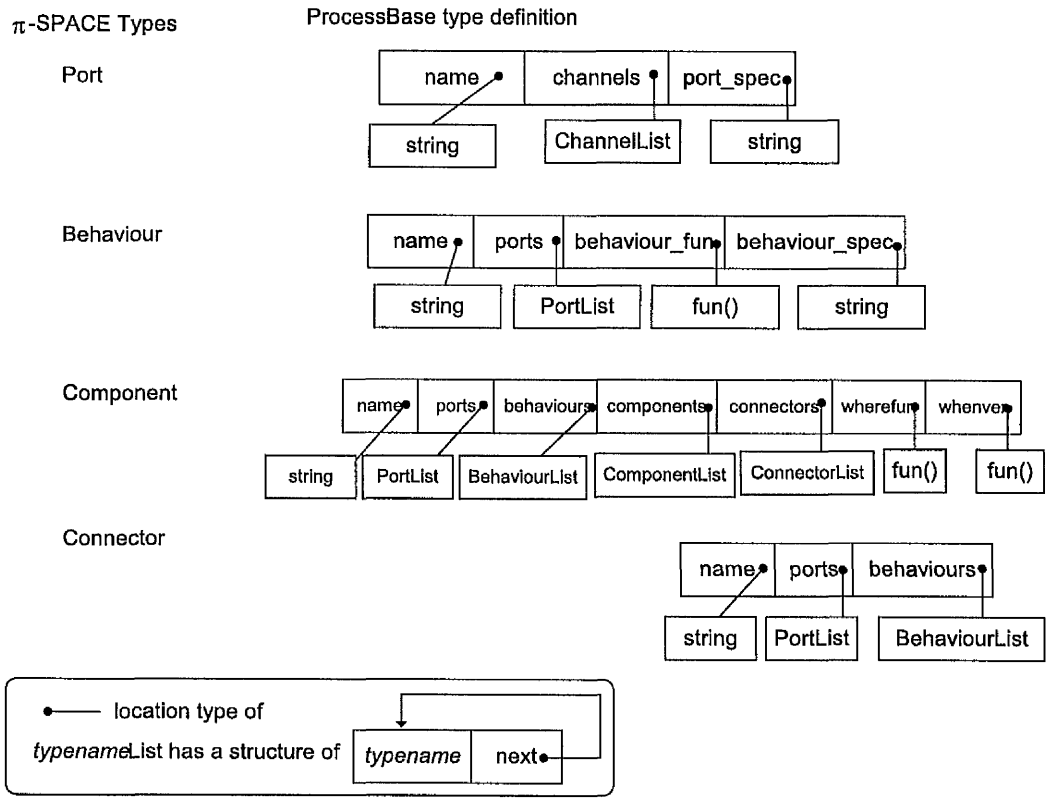

Figure 5.3: π-SPACE types and their associated representations in ProcessBase

Each type will be described in detail in the following list by firstly

describing their definition in ProcessBase and secondly providing the justification of how the elements in the structure are used.

- Channel

  ProcessBase Definition

  This was described in the previous section that describes the mechanisms for supporting the Communication policy.

- Port

  ProcessBase Definition

```
view[name:loc[string];
     channels:loc[ChannelList];
     port_spec:loc[string]
]
```

  A port is made up of a collection of channels. The `port_spec` element stores the π-calculus specification for a Port in string format. This is stored as a string type as a way to retain its original definition which could be used later.

- Behaviour

  ProcessBase Definition

```
view[name:loc[string];
     ports:loc[PortList];
     behaviour_fun:loc[fun()];
     behaviour_spec:loc[string]
]
```

  A behaviour type is defined by the list of ports which can be operated on by the behaviour. `behaviour_fun` is a reference to a implemented function that provides an enactable representation in ProcessBase of the behaviour as specified in π-calculus. `behaviour_spec` stores the π-calculus specification of the behaviour in string format.

- Component

  ProcessBase Definition

```
view[name:loc[string];
     ports:loc[PortList];
     behaviours:loc[BehaviourList];
```

```
        components:loc[ComponentList];
        connectors:loc[ConnectorList];
        wherefun:loc[fun()];
        whenever:loc[fun()]
    ]
```

A component is made up of a list of ports, behaviours, component and connectors. `wherefun` and `whenever` store the references to functions which implement equivalent behaviour in ProcessBase of the *where* and *whenever* operations in π-SPACE respectively.

- Connector

  ProcessBase Definition

```
view[name: loc[string];
        ports : loc[PortList];
        behaviours : loc[BehaviourList]
]
```

  A connector is made up of a list of zero or many ports and behaviours.

- Operation

  ProcessBase Definition

```
view[name:loc[string];
        start_fun : loc[fun()]
]
```

  The Operation type is made up of the `name` of the operation and the `start_fun` which stores the location of the ProcessBase function that provides an enactable format of the Operation. Parameters are not shown in the type definition as they are bound during creation.

- Functions

  - Generators

    There are generator functions for each type and they are usually in the form of:-

    gen<*Component Type*>(<*Component Parameters*>) For example a generator for Component has the following ProcessBase definition:-

```
genComponent <- fun(name:string;
    ports : loc[PortList];
    behaviours : loc[BehaviourList];
    components : loc[ComponentList];
    connectors : loc[ConnectorList];
    wherefun : loc[fun()];
    wheneverfun : loc[fun()] ) -> Component
```

- Utility Functions

  There are various functions for manipulating the data types. A summary of these are shown as follows:-

  add<*Component Type*>(<*Component Parameters*>)

  Adds a component of type *Component Type* to a list specified in *Component Parameters.*

  `getType_string <- fun(x:any)->string`

  Returns the typename of the `x`.

4. Global Control Structure

   Global Data structures consists of all the data structures that are used to store the state of all the entities that are executing within the VM.

   - Data Types
     - Tables

       Each type within the π-SPACE type is store as Tables. Each π-SPACE type has a table which is stored as a binary tree that is identified by its name.

       `ComponentTable`
       `ConnectorTable`
       `PortTable`
       `BehaviourTable`
       `ChannelTable`
       `OperationTable`

     - Process Root

       ProcessBase Definition

```
Process is view [id:int];
    name:loc[string];
    rootComponent:loc[Component];
    components:loc[ComponentList];
    connectors:loc[ConnectorList];
    ports:loc[PortList];
    channels:loc[ChannelList];
    behaviours:loc[BehaviourList];
    operations:loc[OperationList]
]
```

The Process is the root for a specific process model.
The rootComponent element within the Process control structure stores the reference to the root Component of the process model. The Component must be defined within the components element.

- Functions
  (a) Process Enactment Support
    i. The composeComponent operation allows the creation of a composite by composing the set of components and connectors based on the where and whenever specification and executing the resultant composition.
    ProcessBase Definition

```
let composeComponent <- fun(components: ComponentList;
  connector: ConnectorList;
where_fun:loc[fun()]; whenever_fun:loc[fun()])
```

    ii. The decomposeComponent operation provides the converse of the composeComponent operation
    ProcessBase Definition

```
let decomposeComponent <- fun(components: ComponentList;
  connector: ConnectorList)
```

    Both these operations are provided as headers where only a specific composition is generated to test for feasibility and validity of the parameters within the context of the experiment.
    iii. There are also operations to manage the Global Structures. As they are represented as a tree structure these operations

include operations for adding, editing and removing the nodes from *type*Table structure. For example for the the `BehaviourTable` structure the ProcessBase definition is:-

```
rec let addBehaviourNode <- fun(
  newNode:Behaviour;
  root:BehaviourTable;
  overwrite:bool )->BehaviourTable

rec let getBehaviourNode <- fun(
  name:string;
  root:BehaviourTable)->BehaviourTable

rec let delBehaviourNode <- fun(
  name:string;
  root:BehaviourTable)->BehaviourTable
```

(b) Communications Support
These functions provides the ability to manage the global structures that support communication.

Figure 5.4 showing a summary of the global data structures that form the control structures used by the VM to keep the state of all the π-SPACE structures.

5.4.3 Mechanisms to support Dynamic Compliance

In addition to the requirement that a system must support static compliance, the following mechanisms are required to support Dynamic Compliance within the VM layer.

1. Reflective Compiler
   The reflective compiler is derived from the stand-alone compiler that has been written in ProcessBase but packaged as a self-contained function in ProcessBase.
   ProcessBase Definition

```
PPEE_compileString<-fun(program:string)->PPEE_compilationResult
```

The function takes as its input a parameter `program` of type `string` and returns the result of compilation in a structure of type `PPEE_compilationResult`. Without going into the details of the `PPEE_compilationResult` type now, as it will be described in chapter 6, the function returns the resultant generated code if `program` is valid π-SPACE.

2. Upcalls/downcalls
   The downcalls are the invocation mechanim for the compiler. The downcall is a function call to the PPEE_compileString or it can be invoked as part of the compile and go operation of the HyperCode Eval operation. Basic HyperCode operations were introduced in chapter 2 and the π-SPACE HyperCode operations will be described in chapter 6.

   The upcall is the feedback of results from the compiler. As for the result of the Eval operation, it will either be a hyperlink to compiled structure or a compilation error message.

3. Meta-Process
   The meta-process is provided later and also described in chapter 6. As such, the mechanisms that are provided in the VM are just functions that provide hooks to the meta-process model at the layer above the VM layer. All that the functions provide here are message routing mechanisms.

Physical Architecture

In ProcessBase, all the described mechanisms were implemented in libraries. Each library file contains the ProcessBase code which implements the types and functions that make up each mechanism. The layout of the libraries forms the physical architecture that is orthogonal to the conceptual architecture. The list of library is as follows:-

1. `ps_entityLib` - contains all the core type definitions.
2. `ps_commsLib` - contains the variables and functions that implements the Communication Control mechanisms. This includes all the functions described in Communication Control.
3. `ps_procLib` - contains the variables and functions that implements the Process Control mechansims. This includes all the functions described in the section on Process Control Support

4. `ps_managerLib` - contains the variables and functions that implements the Global Control mechanisms.

5. `ps_compLib` - contains the reflective compiler.

6. `ps_codegenLib` - contains the functions that facilitates the code generation phase of the reflective compiler.

7. `ps_utilities` - contains the utilities that can be used in other libraries.

Figure 5.5 shows the physical architecture of the ProcessBase libraries implemented for the πPVM.

Bootstrapping

The approach taken to bootstrap a process is by the use of an Eval operation provided by the HyperCode system. The HyperCode system will be described in chapter 6 but essentially the Eval operation provides a 'compile and go' operation and if the compilation operation is successful, the code fragment is executed and a hyperlink to that enacting fragment is returned. The hyperlink provides a self-contained reference to the code fragment which can be used as a value within the language. For example when we eval a π-SPACE definition of two components, the result of the Eval is two executing component definitions and their references are returned as two hyperlinks. So, for example, we can then use the compose operation to combine them where the parameters to this compose operation are the two hyperlinks to each of the components.

5.5 Criteria for VM Compliance

As PBAM is a CSA tool that was built to support the needs of a Process Modelling System[93, 96] the basic mechanisms that were provided could mostly be reused. This included the basic libraries that support thread control, exceptions handling, persistence, and reflection within the language. Most of the work required was generally on the design and implementation of additional mechanisms for supporting the π-SPACE language.

This point verifies the assumption that csa-based tools, which are built with the ability to support the set of policy needs of the application, will be easier to implement and evolve by requiring fewer changes to the tools.

Part of the VM is also provided in the HyperCode system. The mechanisms that were constructed at this layer are expected to support static compliance. Dynamic compliance for process models are provided by the interaction of the user with the model. The infrastructure to support dynamic compliance is provided by a meta-process which is also provided in the layer above the VM. The HyperCode system and the meta-process will be described in chapter 6. In this manner, the policy for evolution is influenced by the user's interaction with the defined process model.

The definition of Generic Compliance also provides a guide in structuring a system into four different criteria. The result of performing this on the πPVM is summarised as follows:-

1. Number of Layers - There is only one layer that has been added.
2. Required system functions - The following system function mechanisms are required:-
   - Thread control which includes thread creation, deletion and execution.
   - Communication mechanisms, data buffering
3. Method for specifying policy information - Currently specified in π-SPACE.
4. Upcall/downcall, horizontal calls - downcalls are performed via ProcessBase function calls. Upcalls are provided by the use of exceptions and interrupts in the ProcessBase language. Horizontal calls are provided by channels written in ProcessBase which are translated into ProcessBase functions calls that provide mechanisms for channels.

5.6 Model for determining VM Compliance

Components

As a compliant systems must be represented as a set of P, M and $\oplus$, the model to determine a csa-based application will now be applied to the VM constructed. The following lists the set of Policies P, Mechanims M and Binding Rule $\oplus$ that can be realised with justifications.

This chapter has described the set of mechanims that are provided by the πPVM. This was implemented as a set of library functions which provided the

abstractions for the π-SPACE ADL. The π-SPACE language, in particular the mechanisms provided by the π-SPACE language that were described in chapter 4 forms the policy needs that are supported by the mechanisms in the VM. Subsequently, the binding rule that provides the downcalls and upcalls between the mechanisms in the πPVM and its policies are derived by understanding how the policies can invoke the mechanims and get the required feedback from the mechanisms. Downcalls in this layer are in the form of ProcessBase functions and the upcalls are provided in the form of a return value from these invocations. Upcalls are also provided in the form of an Exception which provides the asynchronous form of upcall.

In summary, the compliance model is realised as:-

1. P = π-SPACE language constructs. These are the language mechanisms which were described in chapter 4.

2. M = π-SPACE language constructs that are implemented as library functions in ProcessBase.

3. $\oplus$ = The binding rule maps the constructs of the π-SPACE language to those that are provided by the mechanisms in the VM.

Binding Rule

1. Downcall
   The downcall is a function invocation from the language to the functions that implement the π-SPACE constructs.

   Policy information for the mechanisms are passed as function parameters.

2. Upcall
   Upcall is implemented in the form of a return value from a function invocation. This form of upcall is expected by the π-SPACE language. Another form of feedback which is unexpected is via an Exception call which needs to be caught by the language.

Figure 5.6 describes the architecture of the πPVM in terms of a logical view and its corresponding view from the perspective of a compliant system.

This model is used to the evaluation chapter in order to determine Layer compliance when it is integrated with the language layer.

5.7 Summary

A VM makes it easier for software systems to be implemented and executed on different hardware and at times different operating systems. The CSA tools, in particular the PBAM, were customised in order to provide a proof of concept that a compliant Abstract Machine not only makes it easier to customise a system for an application but also allows customisations that were not possible with tools that are not compliant. The work detailed here also shows the range of flexibility that is provided by a CSA tool.

A model that will be used to determine the compliance of the VM has also been described in this chapter. This model only provides a glimpse into how the mechanisms available at the VM layer can be compliant to some of the policy needs that originated from the definition of the code generation rules for enactable π-SPACE. This model is used later in chapter 7

Policy needs and thus the process model for evolution/change can be defined by the way the user interacts with the underlying mechanisms in the VM. This chapter only decribed the underlying mechanisms of the πPVM that will be used to support the policy needs of the application. The following chapter will describe the interface which the user uses to interact with the system and how the meta process can be captured by a software process framework.

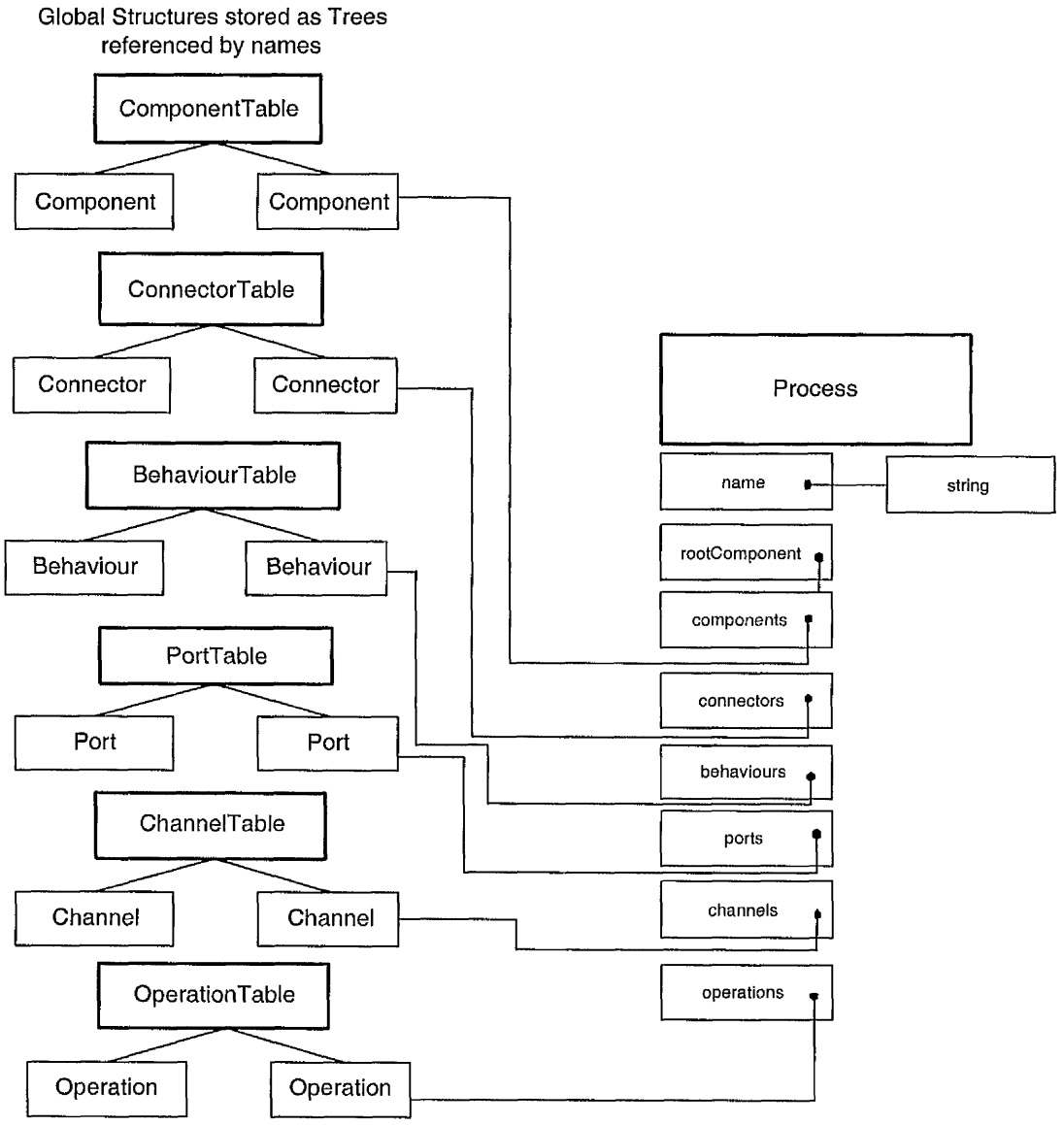

Figure 5.4: Global Control Structures

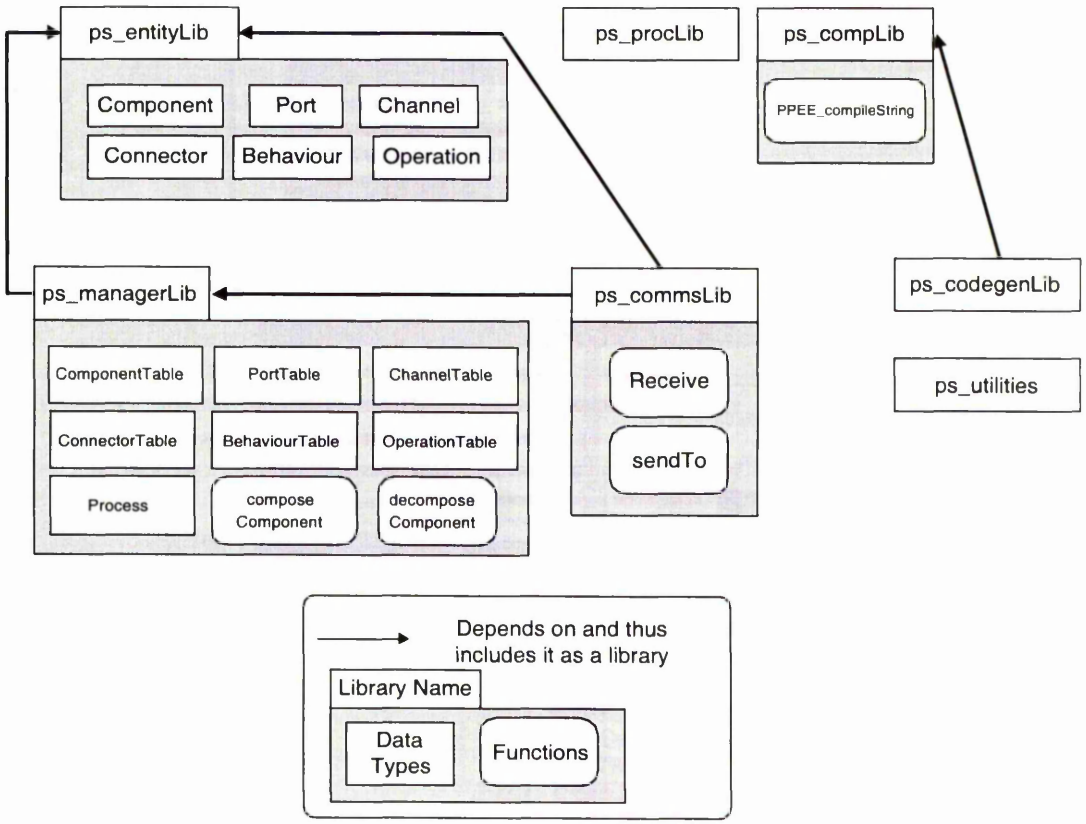

Figure 5.5: Physical Architecture of libraries in ProcessBase

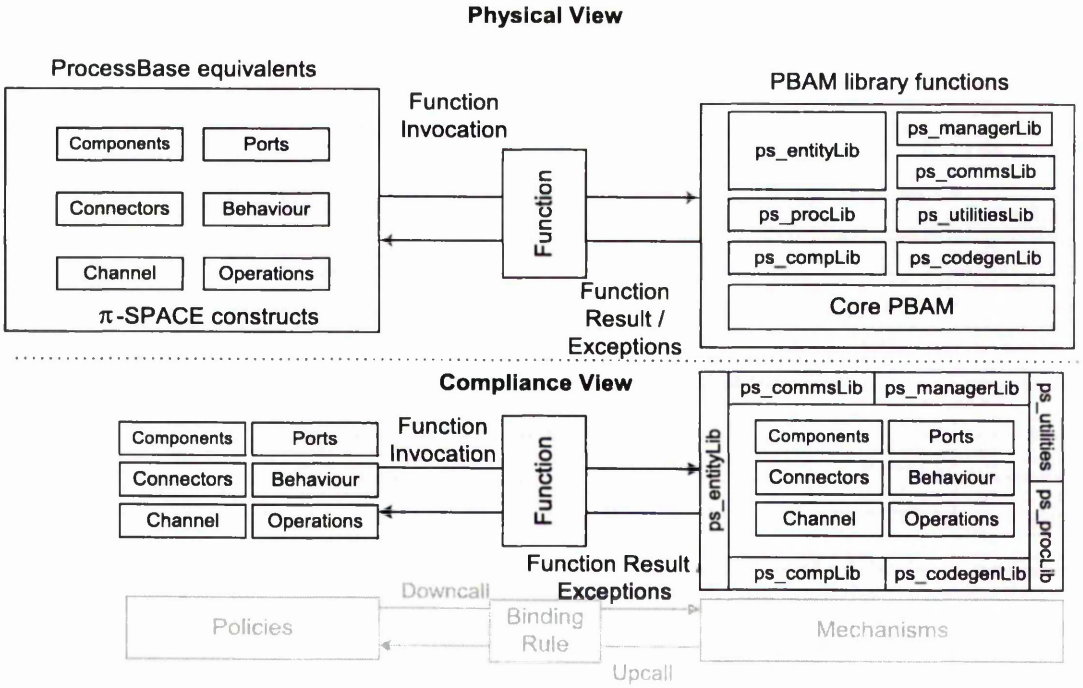

Figure 5.6: Physical and compliant models of the πPVM

Chapter 6

Application Compliance

6.1 Introduction

This chapter describes the design and implementation of the top-most csa-based software layer which utilises the mechanisms of the underlying layers that were described in chapters 4 and 5. This layer provides a compliant layer to support the construction, enaction and management of process models on a PSEE. As this layer essentially forms the interface with the application domain, the chapter starts by describing the mechanisms that were designed and implemented to support the policies required by the application domain of the PSEE.

A PSEE application can be described as interacting with the process domain at two levels. These levels include:-

- An interface to specify and manipulate executing process models
- A meta-process for supporting evolution that is used to structure the resultant models created via the above-mentioned interface.

The development interface is provided by a HyperCode System and the meta-process itself will be based on a generic process framework called Towers[31, 78]. Both layers will be described in detail and, where appropriate, their design and construction. The model of determining compliance will be applied to each layer separately. A description of the work to integrate the HyperCode System and the Towers framework is then provided.

6.2 A π-SPACE HyperCode System

A HyperCode System provides a suitable interface for the development of process models in the π-SPACE language due to its ability to manage and display both the textual representation of static code fragments and their executable equivalent. This allows the representation of both the static and also dynamic definitions of process models in terms of text and hyperlinks respectively. The benefits of utilising such a system as an aid to software development have already been described by Vangelis [105]. In this section a description is given of the work that has been done to develop a HyperCode System for the π-SPACE language. In order to achieve this, the hypercode representation for the π-SPACE language is defined after which the customised HyperCode System to support them is described.

6.2.1 Preliminaries

The conceptual underpinings of a generic HyperCode System are as described by Vangelis[105] and were also briefly revisited in chapter 2. They will be used as a guide for designing and constructing a hypercode system that is customised for the π-SPACE language. Both available HyperCode Systems for Java and ProcessBase, as constructed and described by Vangelis[105], were used as the basis for constructing a HyperCode System for the π-SPACE language. As with other CSA tools, the approach taken was to customise the CSA tools by way of reusing the available mechanisms where appropriate and by extending the set of mechanisms when policy needs are not met by the default mechanims.

6.2.2 Conceptual Model

The conceptual model of the HyperCode System for π-SPACE was largely influenced by the design of the πPVM described in chapter 5 and especially by the policies that the available mechanisms support. The πPVM is essentially a customised PBAM interpreter with libraries to support the abstractions provided by the π-SPACE language. In order to translate the π-SPACE entities into their equivalent ProcessBase entities, a translator/compiler which translated π-SPACE into their semantic equivalent representations in ProcessBase was introduced into the conceptual model. The design and construction of the translator was already

described in chapter 4 but its use within the HyperCode System will be described later.

As the original HyperCode System only operated within a single language, the tasks of the domain operations were to maintain the consistency between the E and R domains. The introduction of another domain, that of the π-SPACE, required the introduction of a new operation. This resulted in a model shown in figure 6.1.

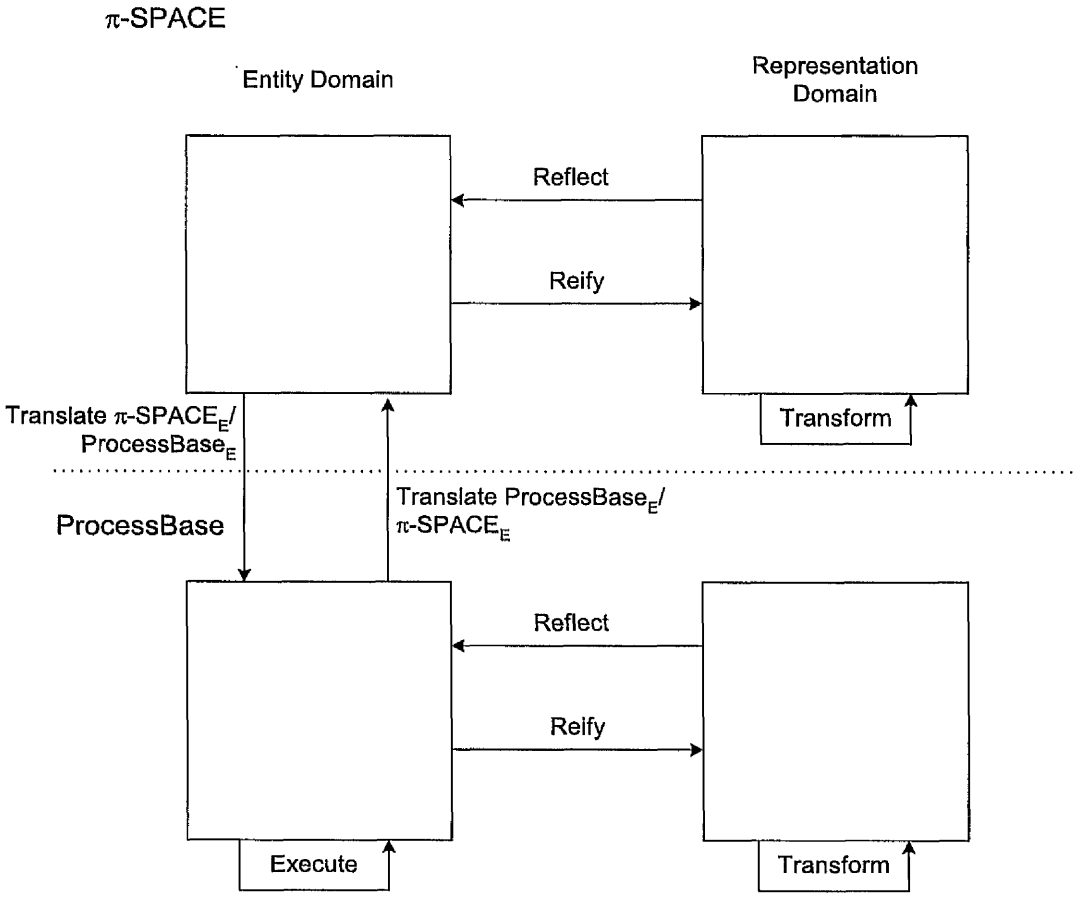

Figure 6.1: The Conceptual Model of the π-SPACE HyperCode System

The details of the conceptual model will be described in detail in the following sections. The descriptions will mainly be in the form of customisations that were made to the HyperCode domain operations and the relevant HyperCode Operations(HCO) that utilises these domain operations.

π-SPACE HCOs	Domain Operations
Evaluate	$\text{reify}_{\pi}(\text{translate}_{p-\pi}(\text{execute}_{p}(\text{translate}_{\pi-p}(\text{ent}_{rep}))))$
Explode	$\text{reify}_{\pi}(\text{translate}_{p-\pi}(\text{translate}_{\pi-p}(\text{reflect}_{\pi}(\text{ent}_{rep}))))$
Implode	$\text{reify}_{\pi}(\text{translate}_{p-\pi}(\text{translate}_{\pi-p}(\text{reflect}_{\pi}(\text{ent}_{rep}))))$
Edit	$\text{transform}_{\pi}(\text{ent}_{rep})$
GetRoot	$\text{reify}(\text{translate}_{p-\pi}(\text{Root}_{ent}))$

The subscript for each operation determines the target language of the operation.

Subscript π is for the π-SPACE language and p is for ProcessBase.

The hyphen(-) is used to described the direction of translation for the translate operation.

The translation should be read as translating a language that is specified by the

subscript on the left to that of the language that is specified by the subscript on the right.

Table 6.1: π-SPACE HyperCode Operations and their Domain Operations

Domain Operations

The four original domain operations, *reify*, *reflect*, *execute*, *transform* were retained for the π-SPACE HyperCode System. These operations were generic enough to support the basic HCO for the π-SPACE HyperCode System.

However, a new domain operation, *translate*, was introduced in order to translate code from π-SPACE to ProcessBase and vice versa. In addition, to provide more detail on the type of language translation operation, a subscript was added to the name of operation. The end result are two operations, the $translate_{p-\pi}$ operation, which is used to represent a translation from ProcessBase to the π-SPACE language and the $translate_{\pi-p}$ operation, which performs the operation of transforming from π-SPACE to ProcessBase.

The introduction of the translate operation seems to fit into the definition of the original HCOs. The end result is summarised in table 6.1 which shows the translate operation fits within the rules of equality that maps the π-SPACE HyperCode operations to the underlying domain operations .

The conceptual mapping of the HCO to their respective domain operations provides a guide for creating a concrete architecture. The concrete architecture however only provides an overview of how the different components of a HyperCode System can be put together. In order to create a HyperCode System, the effects of each HCO on the HyperCode representation are now described.

HyperCode Operations(HCO)

The following list describes the role of each HCO and if relevant the result of the operation on the hypercode representation.

- Evaluate
  The Evaluate operation compiles the textual representation and if it is valid, executes the π-SPACE model. If the result of the evaluation returns a value or type, a link is returned and displayed on the HyperCode Client. This link can then be used by other HCOs.

- Explode
  Explode reveals more information about a particular hyperlink instance. The type of information which is shown after an explode operation depends on the type of hyperlinks.

  Table 6.2 shows some illustrations of the result of explode operation on the different π-SPACE hyperlink types or values.

- Implode - Implode is just a converse of the Explode operation where the the exploded representation is returned back to its simplified representation. This operation can only be applied to a hyperlink view where an Explode operation has been earlier applied. This means that an Implode operation will not have any result at all on a Hyperlink if it has not been through an Explode operation. The intuition is that you cannot implode more than you have exploded with respect to a HyperCode object.

- Edit - This is the editing activity during the writing of code. These activities can range from just writing valid source code text to sophisticated code editing functions such as cutting and dragging and dropping of valid hyperlinks. The range of activity is defined by the facilities available in the user interface tool which is termed the HyperCode Assistant which will be described in section 6.2.3.

- GetRoot - This operation returns a hyperlink to the Root of the Persistent Store. The reason for this operation is to provide a grounding so that hypercode objects can be persistent across development sessions. Any hyperlinks that are not placed somewhere that is accessible from the Root of Persistent will be transient and thus lost after the store is garbage collected.

HyperLink to..	Examples of exploded hyper-links	Values
operation	operation(in[ ], ...], inout[ ],....], out[ ], ...]) { ProcessBase }	Links to parameters with in, inout and out specifiers. Links to ProcessBase representation.
channel	["string"]], [1]	Channels with string literals and integer literals respectively
port	port[ ], ...] {specification}	Links to channels instances and port specification
behaviour	behaviour[ ],...] {specification}	Links to port instances and behaviour specification

component	component
	{
	decl
	ports],...
	behaviours , ...
	}
		Links to declaration(ie value or operations), port and behaviour instances

connector	connector
	{
	decl
	ports],...
	behaviours , ...
	}
		Links to declaration(ie value or operations), port and behaviour instances

composite	composite
	{
	components]: ||
	connectors]:
	
	where
	
	whenever
	
	}
		Links to instances of components and connectors with their corresponding types. There are also links to the instances of where and whenever operations.

where op: attach	attach to	Links to attached channels
where op: replace	replace by	Links to component instances

HyperLink to..	HCR in R
value	
type	

Table 6.2: Effects of the Explode operation on π-SPACE Hyperlink types

6.2.3 Physical Model

Having described the conceptual model and its effects on the representation of hypercode objects, the physical model can now be described. Describing the architecture will provide an overview of the design after which each component within the architecture will be further explained.

Architecture

The HyperCode System is built following a physical client-server architecture where the responsibility is to provide the facilities of an interface to the services that are provided by the server. The client, termed the HyperCode Assistant(HCA) provides the User Interface(UI) frontend which allows developers to specify and manipulate models by using the available HyperCode operations from the HCA. From a csa perspective, the user thus specifies the policies by using the mechanisms that are available as HCOs. Subsequently, the HCOs on the HCA are viewed by the HyperCode Server as the policies that need to be supported by the mechanisms provided by the server.

The HCA and HCS are linked by a communications channel that allows information to be exchanged between them.

Figure 6.2 describes the architecture of the π-SPACE HyperCode System in terms of a logical view that is made up of a HCA, HCS and a communications channel that links the HCA and HCS.

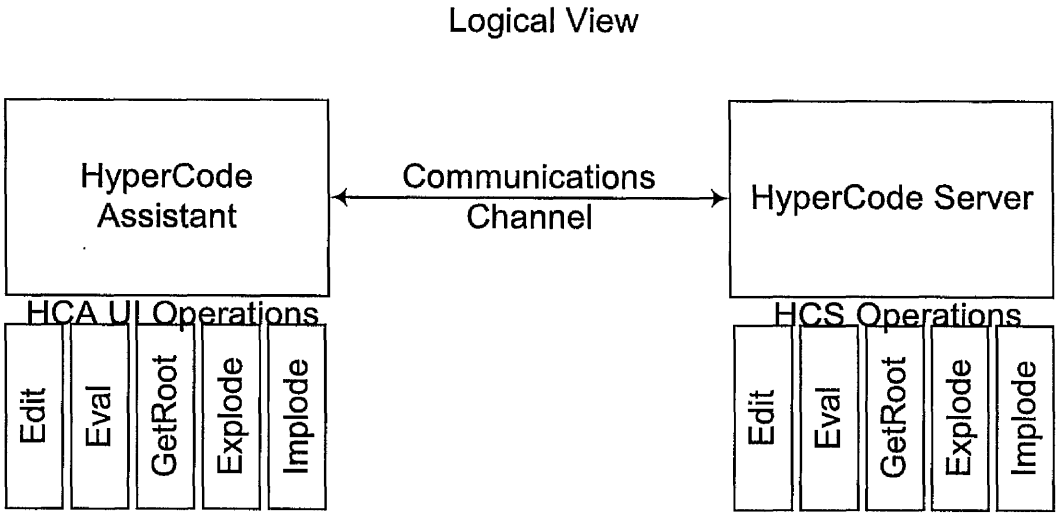

Figure 6.2: The Architecture of the π-SPACE HyperCode System

Each component of this architecture will be further described in detail in the following sections with emphasis on the customisation work that was completed.

HyperCode Assistant(HCA)

The π-SPACE HCA provides the frontend to the HCOs that were described in 6.2.2. The π-SPACE HCA retains all the basic functionality of the original HCA except for the addition of support for compiling π-SPACE code. The UI changes are kept to a minimum through the introduction of a π-SPACE evaluate button. In order to simplify the implementation, the processbase evaluate is still retained and in fact supports the Evaluate HCO for the ProcessBase language. This is possible as at the base level, the π-SPACE Hypercode is compiled into ProcessBase HyperCode.

Figure 6.3 shows a screenshot of the HCA for the π-SPACE HyperCode System illustrating the main features of the client.

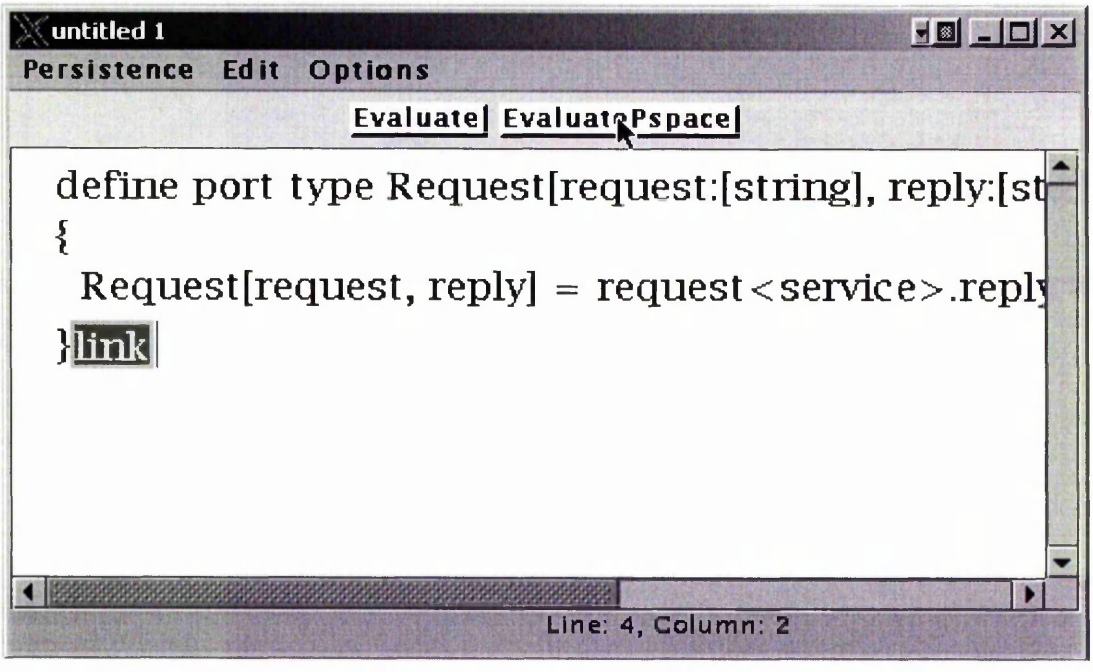

Figure 6.3: A screenshot of the HCA showing the added π-SPACE button

HyperCode Server(HCS)

The HCS listens to the requests from the HCA, processes the requests and responds to the HCA according. The HCS encapsulates the HCO and domain

operations that have been described in 6.2.2 and 6.2.2 respectively. A description of the architecture with references to the original HCS is required to understand the customisations that were required in constructing the HCS for the π-SPACE language.

Figure 6.4 shows the added portions and the data format required to store the extra HyperCode elements for the π-SPACE language.

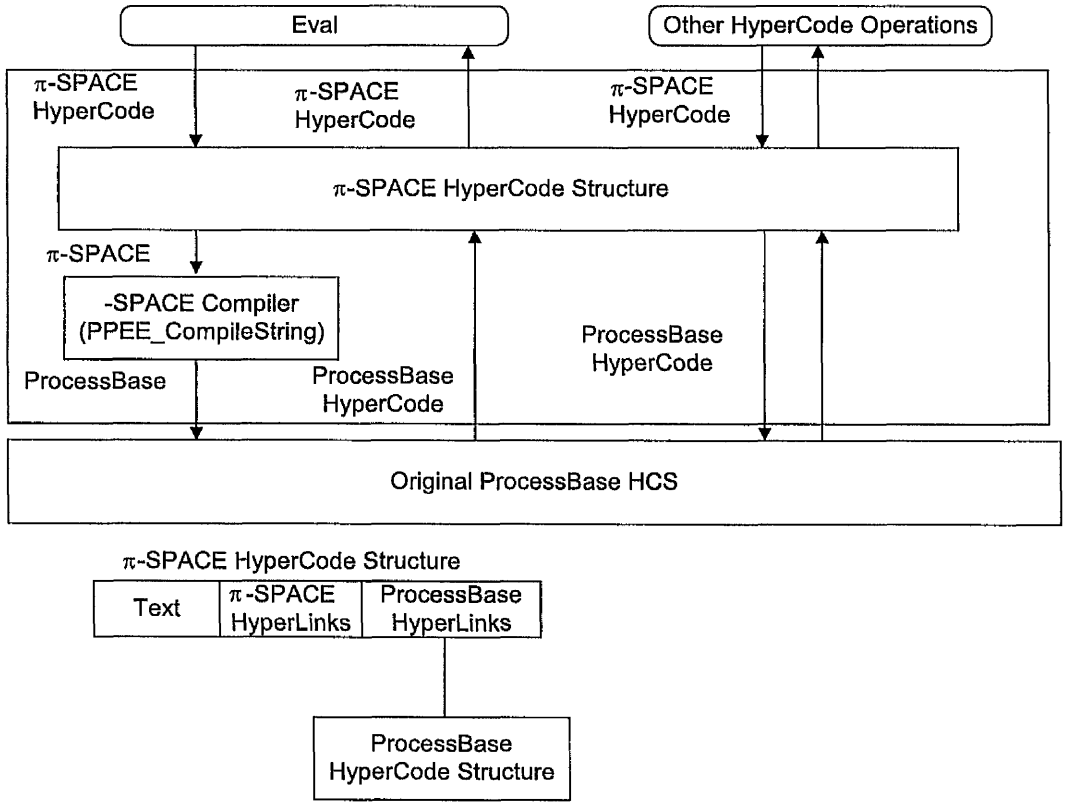

Figure 6.4: The customisations made for the π-SPACE HCS

Communications Channel

The communication channel is implemented as a network socket[77, 88] connection. The communication channel instance is only created when the HCS opens and binds to a default port, and the HCA connects to the HCS port and starts sending data via that port.

A basic request and reply protocol is used to facilitate the communication between the HCA and HCS. This protocol contains a message format that allows

specific HCO selection and their associated parameters to be passed from the HCA to the HCS.

6.3 Determining the Compliance of the π-SPACE HyperCode System

The compliance of the π-SPACE HyperCode System is determined by the presence of a binding rule for each of the policy needs. The binding rule itself must satisfy the basic attributes of a downcall and upcall where policy information can be passed downwards to the underlying layers and mechanism information can be passed upwards back to the policy.

As the HyperCode System has been described in detailed in terms of its conceptual and physical model, it is instructive to apply the model of determining compliance to both models in order to better understand if an abstract compliant model is applicable and useful for describing a flexible system.

6.3.1 Conceptual Model

Components

A compliant systems must be represented as a set of P, M and $\oplus$. The following lists the set of Policies P, Mechanisms M and Binding Rule $\oplus$ that can be realised with justifications.

1. P = π-SPACE HCO, each π-SPACE HCO defines an operation which forms the policy that needs to be satisfied.

2. M = π-SPACE Domain Operations, the domain operations provides the mechanims which is designed to support the policy as defined by the each π-SPACE HCO.

3. $\oplus$ = Rules of equality that matches the HCO to their respective domain operations.

Binding Rule

1. Downcall
   The equivalence rules that map each HCO to their respective domain operations Policy information

2. Upcall
   The equivalence rules that map each combination to their respective HCO

Measuring Layer Compliance

In order to measure layer compliance, the Compliance function Γ can be used. As there are only four policy elements in the set of P, it is clear how the Compliance function Γ is T. As there exists a binding rule $\oplus$ which maps a policy to one or more mechanism(rule of equivalence) for every element p in the set of P (π-SPACE HCOs), the conceptual model is said to be compliant to the needs of the policies.

Figure 6.5 describes the architecture of the π-SPACE HCS in terms of a logical view and its corresponding view from the perspective of a compliant system.

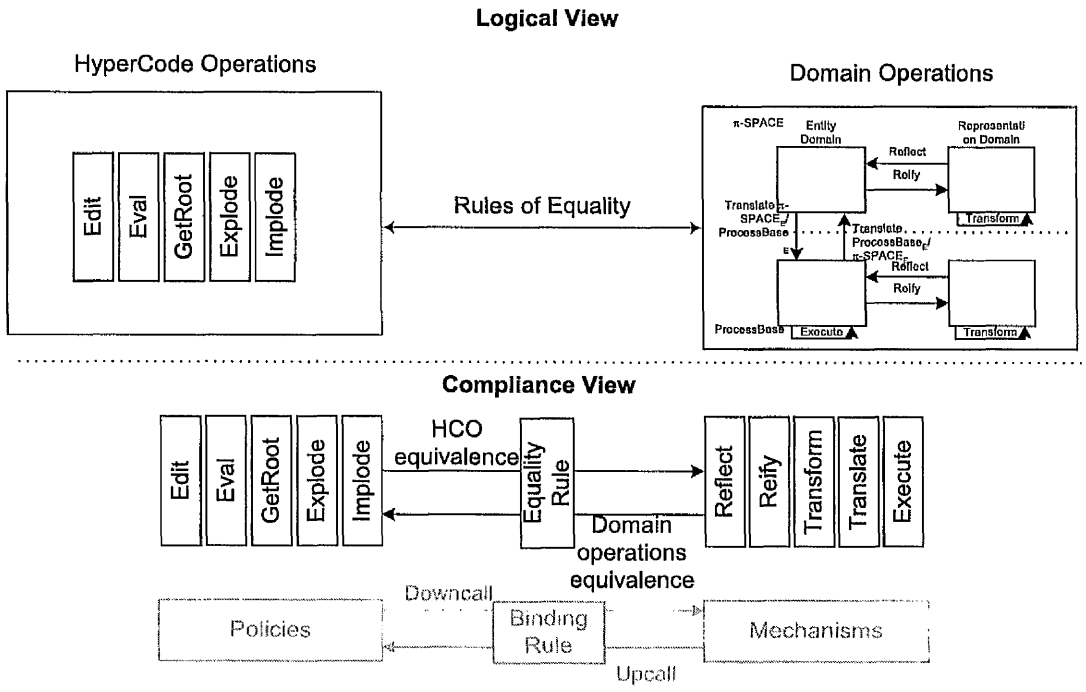

Figure 6.5: Conceptual Model of the π-SPACE HyperCode System as a Compliant Systems Architecture

6.3.2 Physical Model

Having applied the csa-model to the conceptual model, it would be useful to see if it can be applied to the physical model.

Components

Decomposing the HyperCode System in a compliant model view of P,M and Γ resulted in the components as follows:-

1. P = HCOs that are available within the UI and presented by the HCA as a variety of graphical widgets which provides a specific policy.
2. M = HCO operations that are implemented on the HCS. These mechanisms are provided at the server end in response to the requests generated by the policy generated by UI actions.
3. $\oplus$ = Socket Communication between the HCA and HCS

Binding Rule

- Downcall
  This is implemented as a socket request from the HCA to the HCS with a protocol that allows the particular HCO to be identified by the HCS.

  Policy information is in the form of which HCO is invoked and its associated parameter.

- Upcall
  Feedback to the policy is provided in the form of a reply to the socket request for each downcall when connecting via a synchronous protocol.

The protocol that provides the binding rule clearly achieves the two-tuple requirement of an downcall and an upcall.

Determining Compliance

Determining that for all policies there is a binding rule that matches it to the underlying mechanisms.

Figure 6.6 shows how the compliant systems view of the physical view of the HCS is realised.

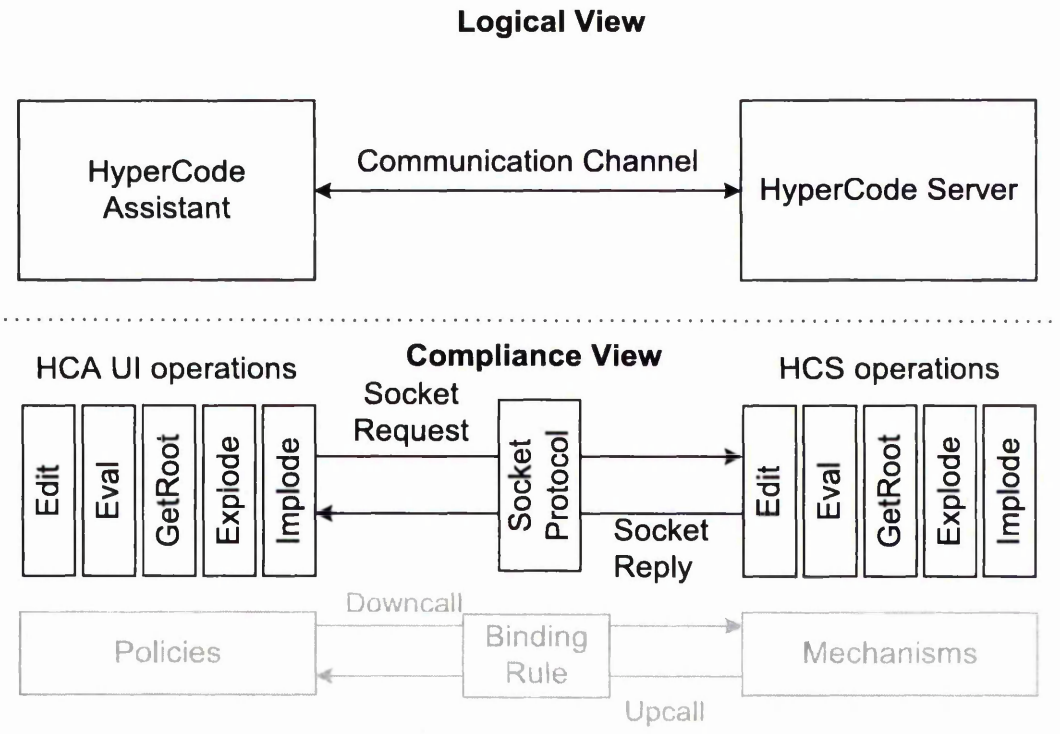

Figure 6.6: Physical Model of the π-SPACE HyperCode System as a Compliant Systems Architecture

Having determined that the interface that will be used to specify process models that are compliant, the meta-process, that is used for structuring the process models, can now be described.

6.4 The Towers Software Process Framework

The Towers Software Process Framework was designed to provide support for dynamic organisations [31]. Essentially the Towers Framework provides the construct in the form of a set of *Nodes* to represent a process model instance and an associated meta-process, Process for Process Evolution(P^2E), which provides support for the evolution of process models through the use of a management process to monitor, manage and install any required changes.

The P^2E meta-process provides a framework that incorporates some form of organizational and management processes. These organizational and management processes includes the monitoring of a process model via its feedback,

detection of the need to change for the monitored process, identification and selection of the types of method now required in response to that need to change and the installation of that method into the process model.

An evaluation to determine the generic properties of the Towers was conducted in a previous experiment[78]. This evaluation tested the generic property by implementing a model of a real-world software process framework, the Rational Unified Process(RUP)[38], by using the Towers framework. The general conclusion from the thesis was that the Towers framework was sufficiently flexible to model a real-world process model due to its approach of viewing processes in terms of the operational process and a meta-process which manages its evolution. A parallel can also be made to the design of the csa model where they are described as only policies, mechanisms and a rule that binds them.

The key strength of the meta-process lies in the way the development nodes were structured as a tower to manage the complexity of multiple products and the multiple dependencies between the products. In addition, each development node is structured such that development processes are separated from, but associated with, the product. This is coupled with a P^2E to support any required changes which the original process was not designed to support. A summary of these ideas were presented in the FEAST 2000 workshop[79].

A Short Description of the Towers

The Towers Software Framework is described briefly to understand its major components. In general, the Towers Framework consists of a set Nodes which are used to represent the operational process and an associated meta-process called, P^2E to manage its evolution. The Nodes are organised in what is termed a Tower. Initially the decomposition might look like a tree but as the Nodes can be decomposed into different views which are orthogonal to other decompositions, this results in a multi-dimensional structure, a Tower.

1. Nodes
    - Specification - This provides a description/definition of the product to be produced
    - Product - This is the resultant product generated by the Develop(see below) operation based on the definition of its Specification
    - Five Operations

(a) Specify - This operation allows changes to the Specification component

(b) Develop - This operation generates a Product that is based on the definition described by Specify

(c) Decompose - This operation decomposes the node into child nodes which themselves are Tower Nodes where each will have its Specification, Product and the five Tower operations.

(d) Build - This operation builds an intermediate product based on the Product in the child nodes. This intermediate product can be used by the Develop process if the Specification contains details for building a product.

(e) Verify - This operation Verifies that the product that was generated in the child nodes is compatible with the Product in the current node.

2. P^2E - A meta-process that supports Dynamic Evolution. It is made up of the following components:-

   (a) Managing - a process that sets the objective to be achieved

   (b) Realizing - organises the installation of a process to achieve the objective set by Managing

   (c) Technology - searches the types of methods and produces the methods(process models) to be used by Realising.

   and the following operations:-

   (a) Install - installs the required changes in the monitored Node.

   (b) Feedback - detects the feedback from the monitored Node.

   (c) Bidirection Information flows between Technology, Realizing and Technology

Figure 6.7 shows the core components of the Towers Software Framework and the interactions between the Nodes and the P^2E meta-process.

π-SPACE description of the Towers Model

There is an implementation of the Towers Model in PML that is enactable on the ProcessWeb system. An exercise was undertaken to describe the Towers in

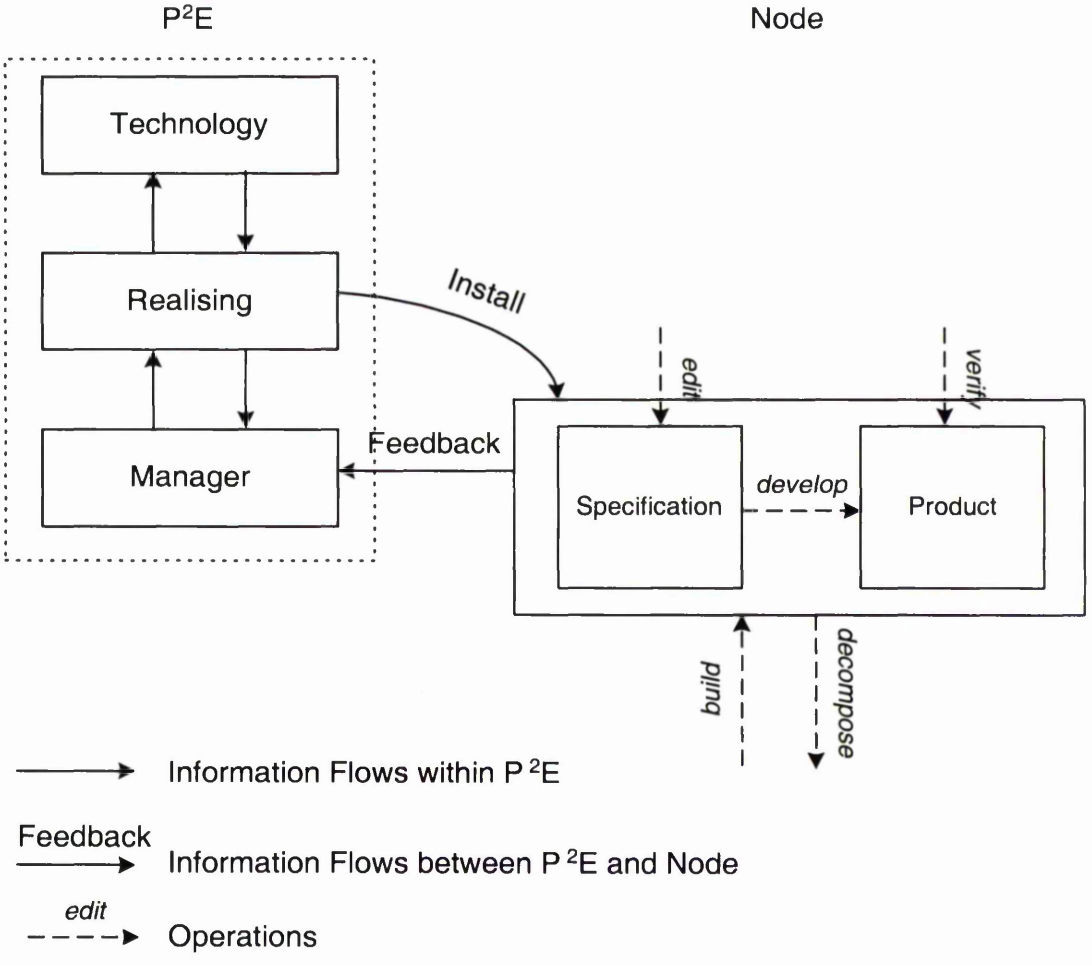

Figure 6.7: The Tower Model which consists of the Node(including Operations) and the P²E Metaprocess

the π-SPACE language. The definitions of the Towers in π-SPACE are included in appendix B.

Determining the Compliance in Towers

Components

1. P = The management processes modelled by the Technology, Realizing and Managing components within the P²E Node.

2. M = The entities, Specification and Product, and the five operations that are represented by the Tower Node.

3. $\oplus$ = The flows of information represented by the Installs and Feedback communication channels between The P^2E and the Tower Node.

Binding Rule

- Downcall
  Install communication channel to support the install operation

  Policy information are in the form of which HCO is invoked and its associated parameter.

- Upcall
  Feedback communication channel to support the feedback operation

Determining Compliance

Figure 6.8 summarises the result of the applying the csa-model on the Towers Software Framework.

6.5 Integration of HCS and Towers

Having described the different layers in detail, the next logical step is to integrate the HCA and the Towers in order to determine if the csa model can be applied to the integrated model. The integration illustrates the first attempt to utilise two different compliant layers in order to construct another compliant layer. The result of this integration will allow a better understanding of how two compliant layers can be integrated and if this integration will continue to result in a compliant layer itself.

6.5.1 Simplifications of Towers

A simplification of the Towers framework was derived from the ArchWare[61, 59] project. This simplified model was more generic and thus allowed an easier integration of the Towers and HCS. The simplified Tower is equivalent to the Tower that was described in section 6.4.

Table 6.3 shows the refinements that has been made in order to simplify the operations of the Tower by integrating the Towers with the HCA.

Figure 6.9 shows the integration of the HCS into the Towers Model.

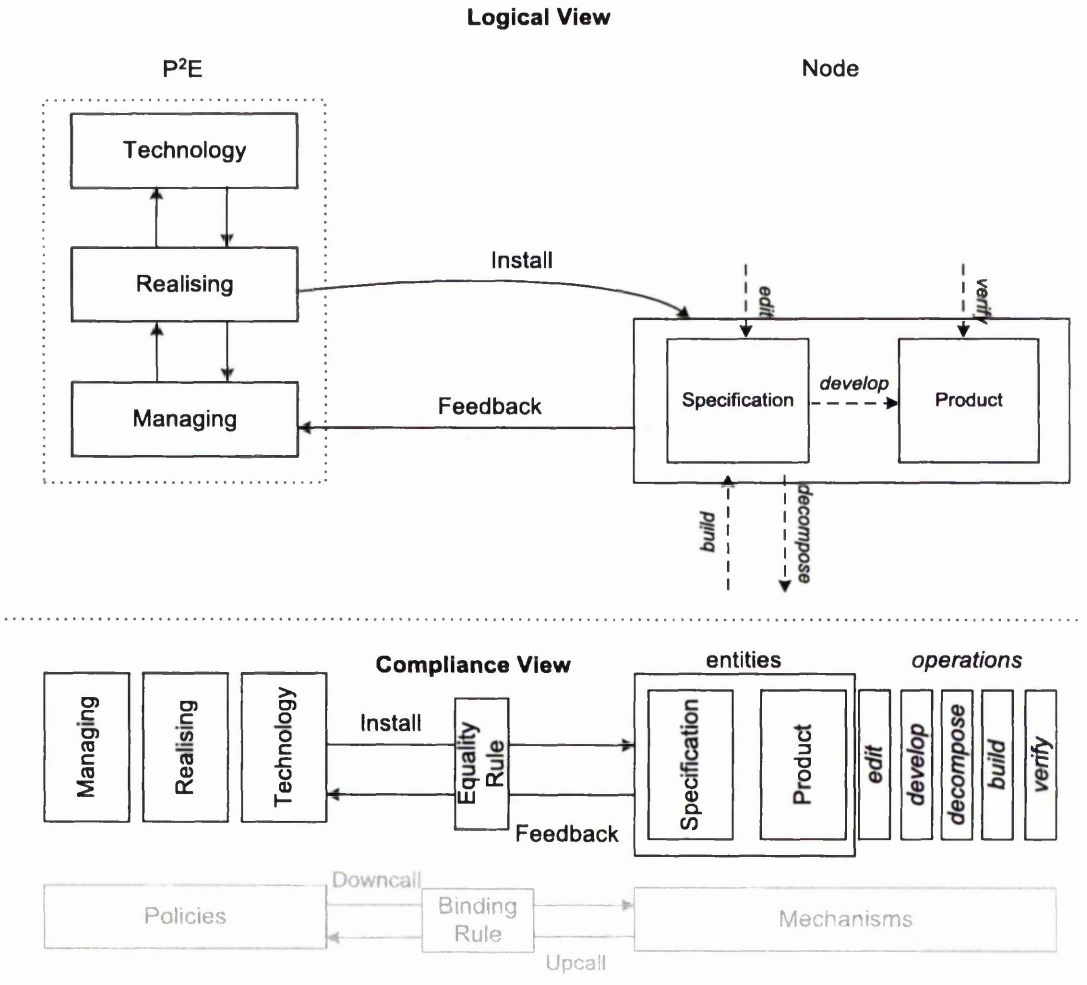

Figure 6.8: The result of applying the csa model on the Towers Software Framework

6.5.2 WebServices

The WebServices[17] technology was used as the communication channel between the Towers operations and the HCS operations.

6.5.3 Determining the Compliance of Integrated HCA and Towers

At each layer, the HCS and The Towers Framework has already been determined to be compliant to the abstract policy needs. The measure of compliance will now be applied to a concrete integrated layer in order to determine if the integrated

Original Operations	Refinement
Decompose	Partition(Towers)
Specify	Refine(Towers)
Verify	Satifies(Towers)
Develop	Evaluate(HCS)
Build(Towers)	Compose operation from VM lib, π-SPACE, directly accessible from the HCS

Table 6.3: Refinement of the Original Tower operations

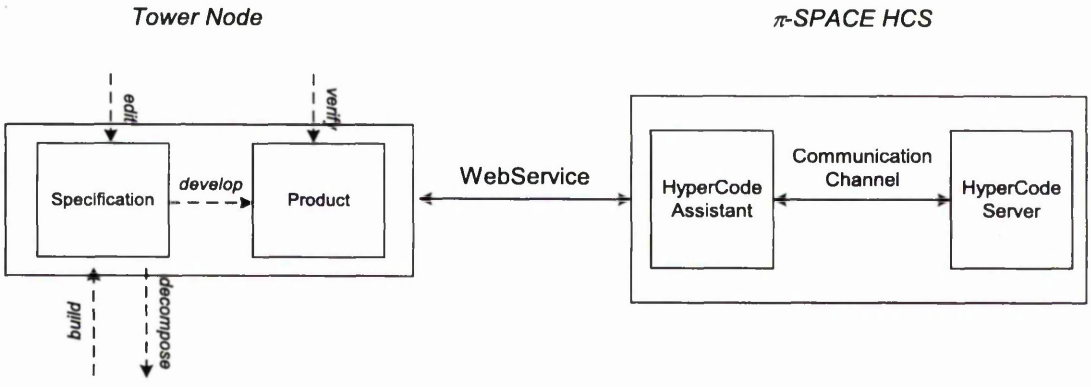

Figure 6.9: The resultant architecture of integrating the Towers Node with the HCS

system can still be deemed compliant. The csa model for determining compliance is applied to the integrated system. As before, a sample implementation of the software was contructed in order to test for the feasibility of constructing the system from the view of compliance.

Components

1. P = Tower Node operations
2. M = HCO provided by the π-SPACE HCS
3. $\oplus$ = WebServices endpoints that support the basic request and reply protocol

Binding Rule

- Downcall
  This is implemented as a WebServices request to a server which is routed to the HCS.

  Policy information are in the form of which Towers Operations have been invoked.

- Upcall
  Feedback to the policy is provided in the form of reply to the Towers Operation

Determining Compliance

Figure 6.10 summarises the application of the csa-model on the integrated Towers and HCA.

6.6 Criteria for Application Compliance

6.6.1 Static Compliance

Static compliance is achieved by the integration of the Towers and HCS. This compliant layer provides the following policies which can be used by any process models.

P^2E

1. Managing
2. Realising
3. Technology

Tower Node/HCS

1. Partition
2. Refine - five HyperCode Operations

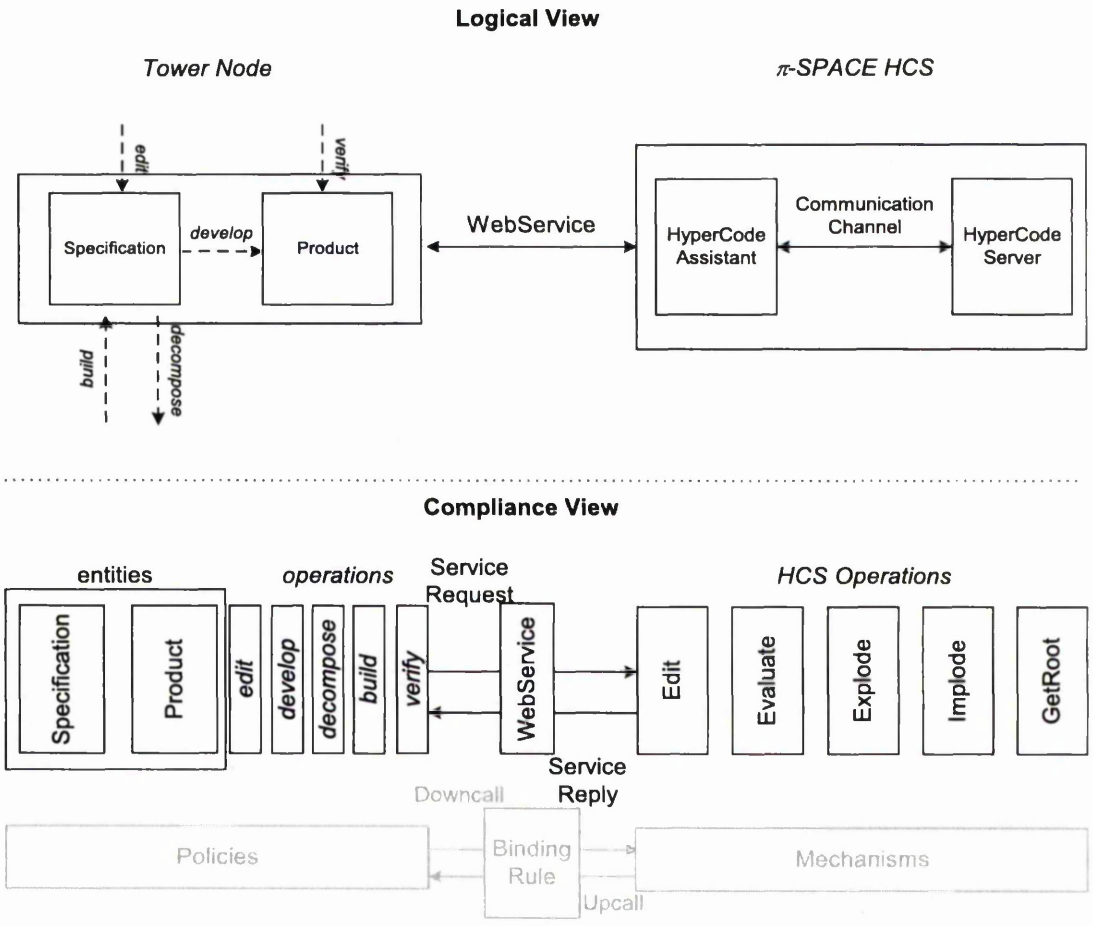

Figure 6.10: The resultant model from applying the csa determination model on the integrated Towers and HyperCode System

6.6.2 Dynamic Compliance

In order to support dynamic compliance, a feedback loop and meta-process were required to detect the need for evolution, enacting the evolution process and for the installation of the new processes into the subject process model. These are provided by the P^2E meta-process. The result of the integration is a meta-process that can now be applied to other software layers.

6.7 Summary

This chapter described the design and the implementation of the final compliant layer in order to achieve tool application compliance. This layer consists of a

HCS for constructing process models and a software process framework, Towers, that allows the structuring of each HyperCode System instance. Both layers were constructed and determined to be compliant after which they were integrated to provide a compliant layer.

The P^2E that operates on Tower Nodes provides an evolution meta-process which allows for the support of process evolution.

The model for measuring compliance was then applied separately to the different layers in order to determine if they can be viewed in a compliant manner. Implementations of the HCS were undertaken in order to verify that the individual components that make up a compliant system have an equivalent enactable form. A basic description of the Towers software framework was completed in π-SPACE. However, the PML model was sufficient as a working meta-process that can be used for our prototype.

Chapter 7

Evaluation of Compliance

7.1 Introduction

This chapter is concerned with the evaluation of the csa model in terms of its suitability for constructing extensible PSEEs which provide for better support of process evolution. In order to achieve this, an example PSEE, which utilises each of the compliant system layers of language, VM, and Development Interface/Evolution Meta-Language that were described in chapters 4, 5 and 6 respectively, was constructed.

The chapter starts by describing the objectives of the evaluation and the prescribed processes for achieving them. This is then followed by a summary of relevant results. In order to highlight the distinction between a PSEE that was constructed on csa layers and that of a non csa-based PSEE, some illustrations of non-compliance are also given. A discussion will then be provided on the approach for re-adapting a non-compliant PSEE into one that is compliant to the new policy needs. An investigation is undertaken in order to explore how a csa-based PSEE can achieve a form of dynamic compliance where it is able to constantly monitor and evolve itself.

7.2 The Evaluation Approach

As there is currently a lack of available concrete data on the effective use of current PSEEs for supporting real-world evolvable software development processes, the evaluation approach will be conducted in a qualitative rather than quantitative manner. It is expected that the results from this qualitative study will contribute

towards futher quantitative studies in this area of PSEEs.

7.2.1 Objectives

Two key objectives were directly apparent at the beginning of the investigation. A third was added later during the course of evaluating the first two.

The first objective is to evaluate the definition and resultant model for measuring system compliance as detailed in chapter 2. In particular the aim is to evaluate if the model is both sufficiently generic and complete for determining the property of compliance for a system. In order to achieve this flexibility, the determination of compliance provided different types of compliance such as layer compliance and system compliance. Evaluation is required to determine if the model is sufficiently complete for it to be useful for system construction.

Each layer has already been determined to be compliant by firstly determining that they can be viewed in terms of a csa model. That is, that the expected policy needs are met by the available mechanisms and then implementing the required mechanisms that will be used by the policies in the layers above. In order to determine if a system has system compliance, however, will require the integration of all the compliant layers into a complete application and then determining if the csa-model can be applied to the complete system. Thus it was necessary to construct the experimental PSEE application to provide a sufficiently concrete example. The layers will be constructed by describing how the policies in the upper layer can effectively utilise the mechanisms that were provided by the immediate layer below.

The second purpose is to compare the degree of flexibility that is provided by a csa-based PSEE with one which was built without compliance in mind. The focus of the investigation will be to deterimine if a csa-based PSEE can provide better support for process enactment and evolution compared to a PSEE that was constructed using conventional software development approaches. Clearly the acid test will be for a PSEE to support process evolution that deviates from what the PSEE was originally designed to support. The assumption is that the process has evolved beyond what the current PSEE is capable of supporting and thus would require more fundamental changes to the underlying core of the PSEE. Of interest would be changes that could not be resolved by changing the model at the level at which it is specified.

A third objective was later derived from the second which was based on the

use of the flexibility of a csa-based system for providing better support of an evolvable process.

7.2.2 Process

The evaluation process involved the use of a sample process model where evolution is required over the course of its process enactment. As a comparison with a current PSEE, Process*Web*[103], which provides a web-based frontend to the ProcessWise[16] PSEE, will be used. Some components of the sample model will be constructed and enacted on both PSEEs in order to evaluate their level of support for evolution. The csa-based PSEE will firstly be evaluated for system compliance. As the determination of system compliance is dependent on each layer possessing the property of layer compliance, the results from applying the csa-model on each layer, that were detailed in the previous chapters, are used in this evaluation. The same process that was used for determining layer compliance is then repeated to form a compliant PSEE which could then be checked for system compliance. This is achieved by repeatedly binding the policies on the upper layer onto the mechanisms in the layer below.

Actual implementations of mechanisms were required to be used as a basis for evaluation as the definitions of csa components, policies, mechanisms and binding rules were themselves in too abstract a form to demonstrate the feasilibity of the model. The construction process, that is the implementation of each compliant layer, and the resultant software that was produced using the process, were used as a proof-of-concept to discover if the compliant attributes of mechanisms, policies and binding rules, do have an equivalent more concrete and executable element. The implementation and the integration is also used as a test for completeness of the csa-model in order to show that the model could be used across different models at different layers of abstraction.

At the highest layer, the policies will be determined by the policy needs of a process model. As an example, a sample basic process model will be described and used in order to evaluate if the πPVM is compliant to the policy needs of the process model. Essentially, this model will provide the top most policies which will utilise the underlying mechanisms that are provided by the PSEE.

In summary, the evaluation process can be detailed as follows:-

1. Determining compliance of each layer

(a) Decomposition of the system into a set of basic csa components - Mechanisms, Policies and Binding Rule.

(b) Determination that the Binding Rule supports the basic upcall and downcall requirement

(c) Determination of the existence of Compliance(Γ) within the layer

2. Determining system compliance

(a) Integration of all the system layers by mapping all policy requirements from the upper layer to the mechanisms provided in the lower layers.

(b) Review of all the Polices and Mechanisms in the Integrated System

(c) Derivation of the Binding Rules from the integration of the compliant layers for the Integrated System

(d) Determination the existence of System Compliance(Γ) within the Integrated System

3. Application of Evolution Scenarios

(a) Description of the W-C (see section 7.4.1) model using the πPVM

(b) Description of some evolution scenarios of the W-C model and illustration of how they can be supported by the πPVM

(c) Provision of some illustrations of evolution that were required by the process model that were not designed into the original πPVM in order to determine if it is able to support this form of evolution. Two scenarios, where the process models might require more fundamental changes, which cannot be catered by specifying the solution in the Process *Web* are shown.

7.3 Evaluation of compliance on integrated layers

Taking a top down approach, the sequence of integration will be as follows:-

1. Application Layer to Language Layer

2. Language Layer to VM Layer

In the following sections, the mechanisms of each compliant layer, Language, VM and Application, will be derived from those that were already described in chapters 4, 5 and 6 respectively. In order to carry out each of the evaluations listed above, the process of integrating the layers involved will be firstly described in some detail. The purpose of the description is to explore and discuss the issues that were faced during integration in order to better understand the role of compliance during integration. After the integration, the determination of compliance is applied to the resultant integrated layer.

7.3.1 Integrating the compliant layers

The determination of compliance has already been applied to each individual Application and Language layers based on the abstract policies that were defined. The determination of compliance will then be applied across the application layers after they are integrated, based on the criteria of evaluation as detailed in chapter 4. The integration process will then be described and then the model for determining the compliance of the integrated layers will be applied.

Application and Language layers

In order to integrate the Application and Language layers, the mechanisms of each layers are listed. The mechanisms at the Application Layer will form the policy needs which the mechanisms at the Language layer will need to support. For the Application Layer, the mechanisms that are available are in the form of an integrated π-SPACE HyperCode System and the P^2E Meta-process. The integrated mechanisms that are available at the Application layer are thus:-

1. HyperCode Operations
   (a) Evaluate
   (b) Implode
   (c) Explode
   (d) Edit
   (e) GetRoot
2. Towers Operations which are the integrated operations available in the P^2E and the Towers Node.

(a) Specification

(b) Product

(c) Five Operations - Specify, Develop, Build, Verify, Decompose

From the viewpoint of the Language layer, these Application layer mechanisms are the policy needs that the Language must support.

The mechanisms provided by the language, in terms of the π-SPACE abstractions, supports the policy needs of all the Tower Operations. This is demonstrated and described in π-SPACE in appendix B. The binding rule is formed by realising the model and then programming it in a process modelling notation. The binding rule is thus described informally, where the downcalls, are all implicitly derived from a programmer's idea of the mechanisms that are required to support the components and operations of the Towers.

The mechanisms to support the HCOs however, are not provided at the Language layer. The reason is that the mechanisms provided by the language are not able to support the HCOs. The actual mechanisms to support the HCOs were implemented within the VM layer.

Language and VM layers

The integration of the Language and VM layers follows a similiar process to that of the previous integration of the Application and Language layers. The mechanisms of the Language layer will now serve as the policies for the VM layers. In order to describe the process, a recap of the mechanisms are provided.

At the language layer, the mechanisms provided by the π-SPACE language are the types and process abstractions available. The types are in the form of *Primitives* which includes **Names** and **Channel types**, and **Aggregates** which consists of the **Port**, **Behaviour**, **Component** and **Connector** types. The three types of process abstractions are in the form of the π-calculus expressions for specifying the behaviour and constraints within each of the types, the operations and annotations which provide programmable syntax and semantics and the evolution operators which allows the specification of events and their response behaviour as specified by the π-calculus expressions.

As the mechanisms provided by the VM are at a lower level, in the sense that it is more concrete, the mechanims described are in terms of libraries of functions

and data structures. VM mechanisms consists of *Libraries* which provide facilities for **Process Control** and **Communication Control**. The data structures provided are abstractions for each of the π-SPACE **types** and a **Global Control** data structure for keeping the entire state of the VM.

Having described the mechanisms, the process of integrating them will now be described. In general, the process involves the mapping of the policy needs from the Language layer onto the mechanisms provided by the VM layer. The term 'mapping' is suitable as it is a sufficiently generic term that indicates a relation between the mechanisms and policies between the layers.

7.3.2 Determining Compliance

The integration process was generally straightforward as the layers had been designed to be compliant in the first place. In fact the csa-model served as a reference model for decomposing systems such that during the implementation, most policies are supported by mechanisms. There was only one exception where this was not true. The policy needs of the HyperCode System were not completely met by the available mechanisms in the language. However, this was due more to not understanding how the original HyperCode System was designed and implemented. This characteristic is something which is common during the software construction process. The HyperCode policies required access to some VM properties which the language did not provide. Having a language that supports active compliance did help in this case as it allowed the use of VM mechanisms from the Application layer.

The determination of the csa-model is applied to the integrated model and the process of determining compliance is performed on it. The result of the integration and compliance determination is show in figure 7.1.

7.3.3 Summary of findings

The integrated system forms the csa-based PSEE which was used to understand how a csa-based PSEE can better support the deviated form of evolution. Some discussions of the findings during the course of integrating and applying the csa-model on the integrated PSEE application is required. The discussions involve the use of the csa-model to guide the course of the integration.

The csa model can be viewed as sufficiently complete as it is able to model

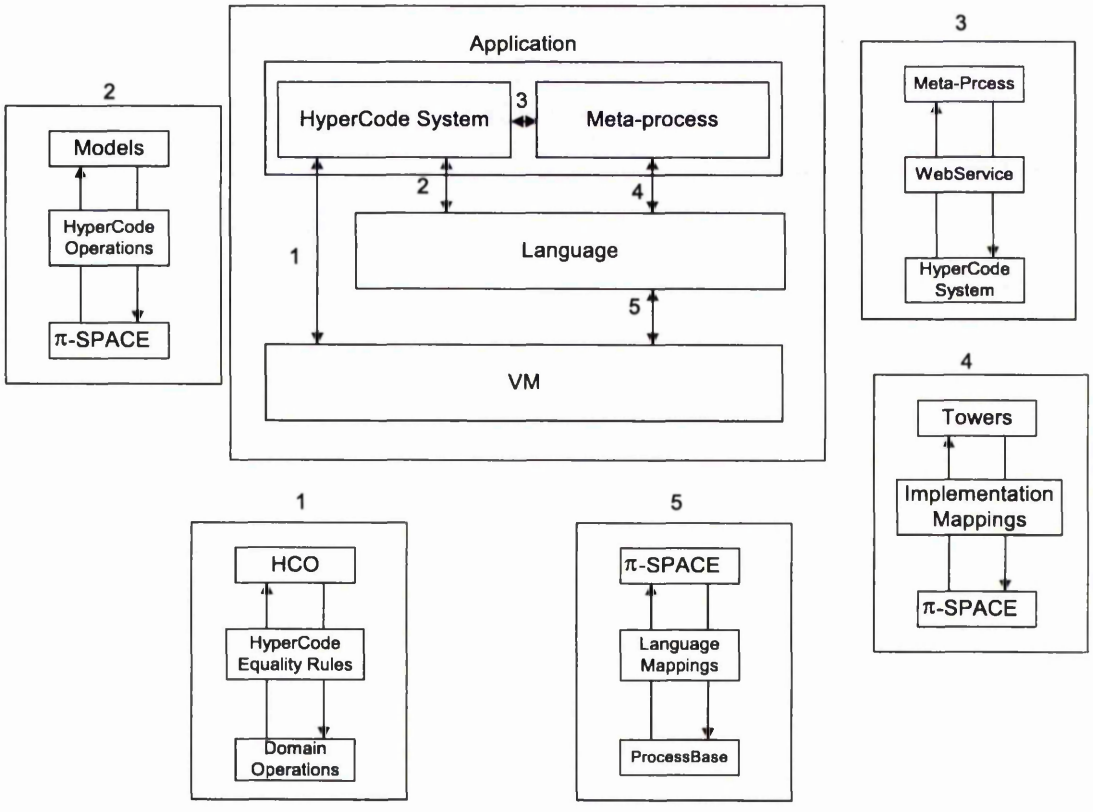

Figure 7.1: The resultant Integrated model of Application, Language and VM Layers

software layers at different levels of abstractions. It is also not unexpected that the lower software layers that are closer to the hardware have more well-defined policies and mechanisms, as there has been a lot of research completed in constructing these layers. The higher layers are more abstract in nature and this resulted in the binding rule being quite abstract with different possible interpretations. During the integration exercise, certain interpretations of the bindings had to be assumed. The approach was to ensure that these abstract layers could easily accomodate change and to introduce a meta-process to manage the changes required.

7.4 Evaluation of a csa for a PSEE

The integrated software application now forms the prototype PSEE which can now be evaluated for the deviated form of evolution. In order to achieve this, a

sample process model was used to determine the application policies that need to be supported by the underlying mechanisms that are provided by the PSEE.

Some illustrations of evolution will be described later that provide a better understanding of the type of evolution support that a csa-based PSEE can provide.

7.4.1 A Sample Application Process Model: The Writer Checker(W-C) Model

The Writer Checker(W-C) process model[18] is an example process model that was designed and constructed in order to study the most basic representation of a simple process. The intention of making it simple was driven by the need to study the evolution needs of such a model and thus enable the testing of the evolution support abilities of an environment to be conducted.

Description of the Model

As the name implies, the W-C model consists of two components, a component called *Writer* and a corresponding component that is labelled as the *Checker*. A communication link connects both components together which allows the components to send messages between the two components.

The behaviour of the Writer is to write a message, and then send it to the Checker to be verified. The content type of the message can be, for example, a piece of code or just a report of sorts. It can be assumed that the content does not matter as the assumption is that the Checker will know how to check the contents. After sending the message, the Writer then waits for the response from the Checker and makes the refinements that were suggested by the Checker. The cycle continues again until the Checker's response is that there is no need to make any changes to the content.

The behaviour of the Checker is to check the contents that have been sent by the Writer against some criteria which the Writer might or might not know and to then provide the result after performing the checking.

This model is surprisingly simple but it embodies the core characteristics that are present in most process models. Some key issues can be summarized as follows:-

1. Each component has some behaviour that can function independently of the other

2. There are interactions between the components

3. The interactions defines the dependencies between the components, which creates the external influence on the behaviours of the process model

This model can also be considered as a client-server model where the Writer is the client that submits requests to the services provided by the server and awaits the response from the server. In order to keep the model simple, the following W-C cycle can be defined. Writer writes message. Writer sends message to Checker. Writer waits for reply indefinitely. Checker is always in the waiting state unless it receives a message from Writer. Checker checks the document and produces a response to Writer. Writer receives a reply from the Checker and decides if it needs to write another document. The cycle is repeated if the decision is to write.

The enactment of this process model is very much like the execution of a simple program if no changes are required to the model for it to be useful. However, changes are bound to happen and this results in a few possible evolution scenarios which will be described to illustrate the complexity of process evolution.

Some scenarios of evolution are:-

1. Evolve from one W to one C to one W to many C

   - Description
     There are now more Checkers for that one Writer. Writer will now send all the writings to multiple Checkers and await for the replies from the Checkers. This change will result in a few interesting scenarios.
     For example, should the writings be sent to all the C when W submits. Or can W send to only a few of the W or even one of them?

2. Evolve from one W to one C to many W to one C

   - Description
     This scenario is similiar to the previous evolution scenario except that the situation is now reversed where there are many Writers that can submit their writings to one Checker.

3. Evolve from one W to one C to many W to many C

- Description
  The situation in this scenario now is rather like a composition of the previous two scenarios.

This list of evolution scenarios are just an indication of how complicated evolution can become even though the original process model is perhaps the simplest possible.

Figure 7.2 shows the illustrations of the W-C model and the evolution scenarios described. The diagram also shows a Switcher which is introduced as a

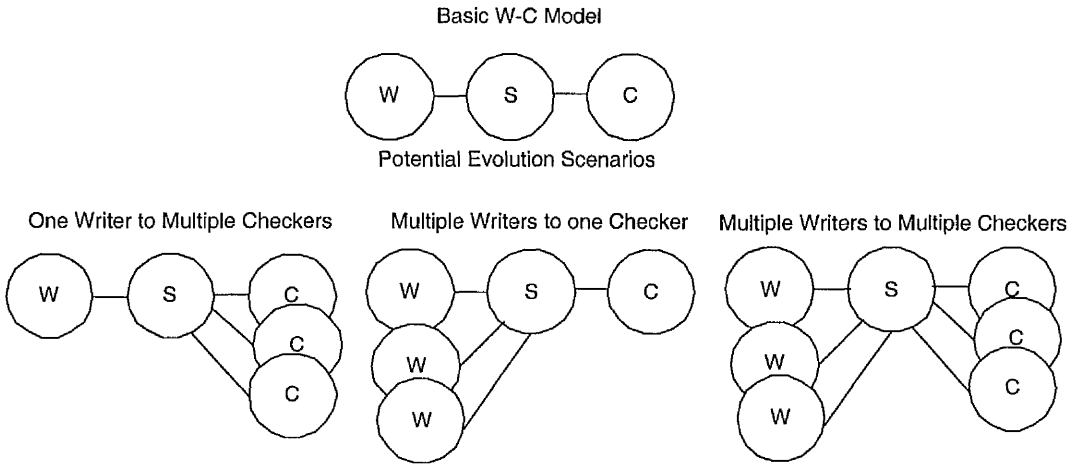

Figure 7.2: The W-C model and some illustrations of W-C model evolution

connector between the Writer and Checker. The Switcher functions as a routing interface between the Writer and Checker.

7.4.2 Comparisons of Evolution Modeling and Support

Comparisons are now made with reference to what is possible using the Process *Web* PSEE. An implementation of the Writer Checker model in Process *Web* is detailed here[18].The types of evolution that are looked into can be classified into designed evolution, where the types of evolution supported have already been catered for within the language, and deviated evolution, where the evolution is not catered by the language due to earlier assumptions made on the concept of a process.

Designed Evolution

This is a type of evolution where the support is already designed and thus catered for the PSEE. An example of this is the 'if then else' expression where the conditions for the evolution and the prescribed process in response to the condition are specified. This type of evolution is generally well known where the response to the condition is also well defined. Designed Evolution can thus be formalised and be well supported by current PSEEs. As this is well supported within Process *Web*, a description in π-SPACE is provided in order to illustrate how the π-SPACE language supports this form of evolution.

π-SPACE description of W-C Model

The W-C Model will now be described in π-SPACE in order to show how it can be specified in the language. These definitions are based on the paper on π-SPACE[18]. An illustration is also provided to show the π-SPACE specification for describing the evolution scenario, where the W-C model is evolved to support multiple Writer Instances.

The W-C model consists of three components which are defined in π-SPACE as follows:-

π-SPACE model of Writer

```
define component type Writer
{
  port require_check: Request[receive,send,module, reply]||
  behaviour write: Write[require_check]
}
```

π-SPACE model of Checker

```
define component type Checker[supply_check:Reply[...]
{
  port supply_check: Reply[receive,send,module, reply]||
  behaviour check: Check[supply_check]
}
```

π-SPACE model of Switcher

```
define connector type Switcher[caller:Reply[...],
```

```
   callee:Request[]]
{
  port caller: Reply[...]||
  port callee:Request[...]||
  behaviour switch:Switch[caller,callee]
}
```

The W-C model is composed of the three components where the Switcher ports are connected to both the Writer and Checker components.

π-SPACE for composing the W, C and S.

```
compose WSC
{
  W1: Writer1||
  S1: Switcher1[caller,
                          callee]||
  C1:Checker1
  where
    attach W1@request_check to S1@caller,
    attach C1@supply_check to S1@callee
}
```

Specifying designed evolution

π-SPACE for composing and decomposing the W, C and S to generate a new set of C and S instances to cater for a Checker and Switcher that deals with multiple W instances.

```
compose WSC2
{
  decompose WSC||
  C2: Checker2||
  S1:Switcher[caller, callee]||
  C1:Checker1
where
  replace C1 by C2,
  recompose(W1, S1,C2)
}
```

The construction of the model using the π-SPACE HyperCode System is illustrated in figure 7.3 Figure 7.3 shows the illustrations of the W-C model and the evolution scenarios described.

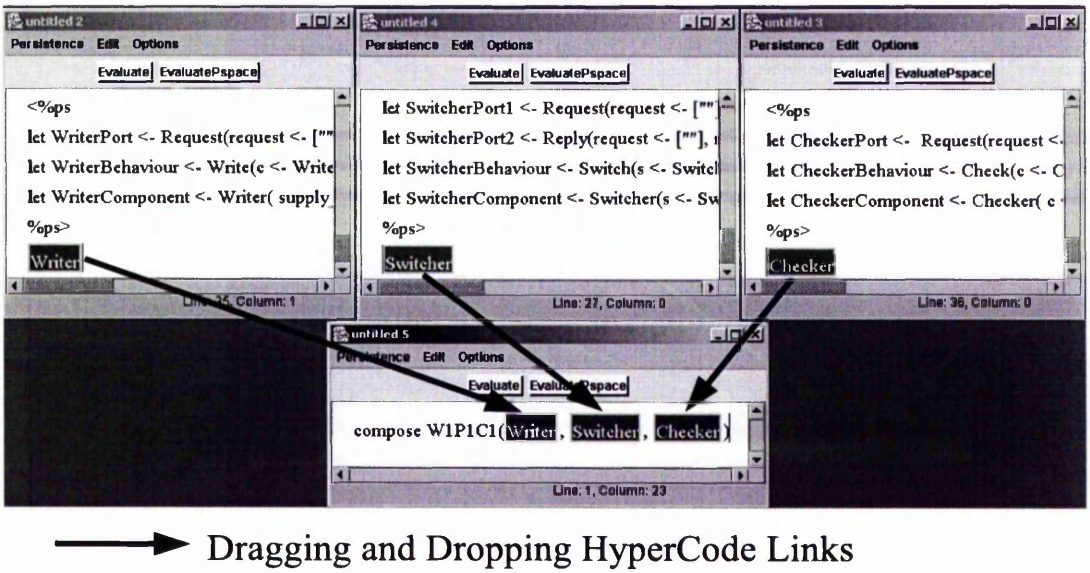

Figure 7.3: Construction of the W-C model using the HyperCode System

Scenarios of Deviated Evolution

The evolution scenarios reveals that even though a system can be designed to be as flexible as possible, there will always be a limit to the flexibility of a systems architecture. A PSEE with static compliance will support all forms of evolution that conventional PSEEs were designed and implemented to support as was clearly illustrated by the evolution example above. However, only a PSEE which supports active compliance will be sufficiently flexible to cater for the types of evolution which demand more fundamental underlying changes to ensure that the process model in the PSEE is still compliant to the real-world model. In this section we explore and detail the possible forms of deviated evolution which cannot be designed and thus supported by implementing them using the mechanisms provided by, for example Process *Web*.

The main types of deviated evolution appear to be caused by changes that cannot be catered for by making changes within the same level as that which the application is executing. The consequence of this is that the evolution itself cannot be detected, managed and resolved at the same level as that at which

the PSEE language is enacting. This type of evolution requires a greater 'transparency' to the available underlying mechanisms.

Two types of evolution that were briefly detailed in a paper presented at the Ninth European Workshop on Software Process Technology[80] are illustrated with reference to how a csa-based PSEE can be better customised to support the deviated form of evolution.

- Policy changes which require feedback from the scheduler and the ability to select the best scheduling scheme
- Communication abstractions

7.5 Non-Compliance

When the underlying mechanisms do not provide sufficient support for the policy needs, or there is no valid binding rule for the policy to the underlying mechanisms, then the software layer is deemed to be non-compliant. The following sections detail the reasons why they were non-compliant and the approach to making the system more compliant to the needs of the application.

7.5.1 Communication model

Non-Compliance

This is a mild form of non-compliance in that it is more of an optimisation for implementing the communication support mechanisms at the VM. The original mechanisms for buffering within the communication model were currently implemented in ProcessBase. This is clearly sufficient for the prototype application, however, there were some mechanisms provided by the Operating System for these functions. This is sufficient but the communications model is improved if it is implemented at the level where the mechanisms can be modeled in a cleaner fashion. Clearly this non-compliance is a result of the duplication of similiar mechanisms across the system.

In addition, the initial design and implementation were determined to be compliant as the implementation assumed that the underlying networking mechanisms were unavailable. The mechanism was later introduced when we utilised

the ArenaOS for some experiments. This provides a good comparison with Linux which is an OS which is non-compliant in this case.

Making it Compliant

In order to make it compliant, the functions in ProcessBase which implemented the communication mechanisms are still retained, thus ensuring that the bindings are retained. The only changes required are to bind these functions onto the actual mechanisms that are available at the Operating System level. With the ArenaOS, some work was undertaken to customise the Networking Manager. After which, some initial work was required to expose this interface to ProcessBase. In this case, we utilised one of the core communication opcodes and customised the parameters that invokes the mechanisms provided by the Networking Manager.

7.5.2 Thread Control model

Anderson[5] described how kernel threads and user threads each have their own issues and thus they are provided as a type of abstraction over both in order to resolve these issues. The issues that were described are similiar to those that were faced by the example process models. The only difference is that our abstraction is now provided by the binding rule in a compliant model.

Non-Compliance

In some scenarios, the default in-built threads scheduling scheme did not provide sufficiently fine-grained thread control facilities that were required by an example process model. Using a non-compliant OS Linux did not allow a process model to have access to different thread scheduling schemes which are more relevant to the process model.

Making it Compliant

An attempt was made, in order to resolve the lack of thread scheduling control by substituting the non-compliance OS with one that provides this facility. The work of relating the importance of thread scheduling mechanisms being made available as a form of feedback for process models are addressed in a chapter

of a book[30] with the tentative title of "The Impact of Software-Architecture Compliance on System Evolution".

An implementation scheme for running the model on the ArenaOS was designed. This provided more control to the process model for accessing and specifying the types of scheduling schemes that are available from the scheduler. The threads can thus be changed and fine-tuned by the process model. In this experiment, the change was done manually as a proof of concept. Subsequent to the work reported in this thesis a dynamic loader for the ArenaOS has been developed[10]. Its evaluation within a complete implementation has not been undertaken but it is clear that it should be quite straightforward.

Figure 7.4 contrasts the non-compliant and compliant architecture for thread scheduling.

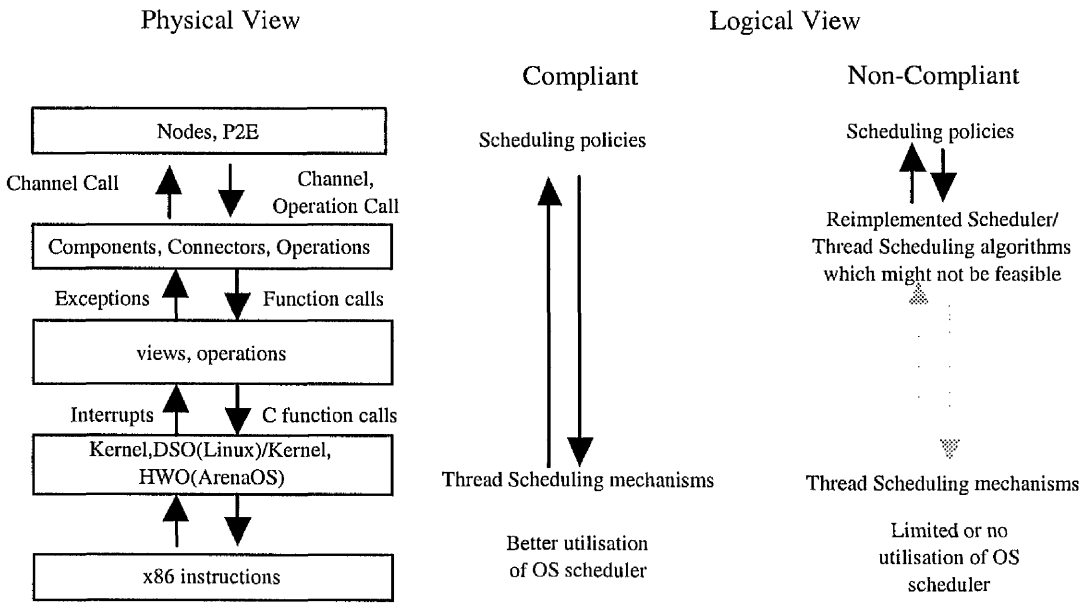

Figure 7.4: A Compliant Architecture view of the Thread Scheduler mechanism

Both these scenarios clearly demonstrate that a csa-based application is able to support the deviated form of application by allowing the underlying mechanisms to be exposed to the language layer. Essentially this allows all the underlying mechanisms to be used to support the current expected policies that have been pre-defined and the new policies that might appear as the real-world process domain changes.

7.6 Summary

The evaluation for compliance for each layer was derived from the definition of a compliant systems provided in chapter 2 and the criteria that were relevant to each level that were as described in chapters 4, 5 and 6.

A complete PSEE, the πPVM, was constructed progressively from each compliant systems layer. The compliance measure model was then applied to the integrated system in order to determine system compliance. A simple application process model was introduced and certain evolution scenarios were described to evaluate the static compliance and dynamic compliance of the πPVM.

Some illustrations of deviated evolution which resulted in non-compliance were discovered when the process model required more fundamental changes to the underlying available mechanisms. These were detailed and understood after which the approach to make them compliant was described. The existence of non-compliance reveals that even when each layer is compliant, the integration of the compliant layers might result in a non-compliant systems architecture. The use of compliant layers however, allowed these issues to be resolved by a technique that can best be described as exposing underlying layers.

The claim that a compliant systems architecture is able to provide a more flexible architecture for constructing a PSEE was also evaluated. In order to substantiate this claim, two illustrations where evolution would require a more fundamental change within the PSEE were shown. Even though it was not an exhaustive test, the experiment demonstrated the simplest case for evolution which should support most, if not all, forms of evolution.

The final model of the specific compliant model that can be derived from the Generic Compliant model, is shown in Figure 7.5

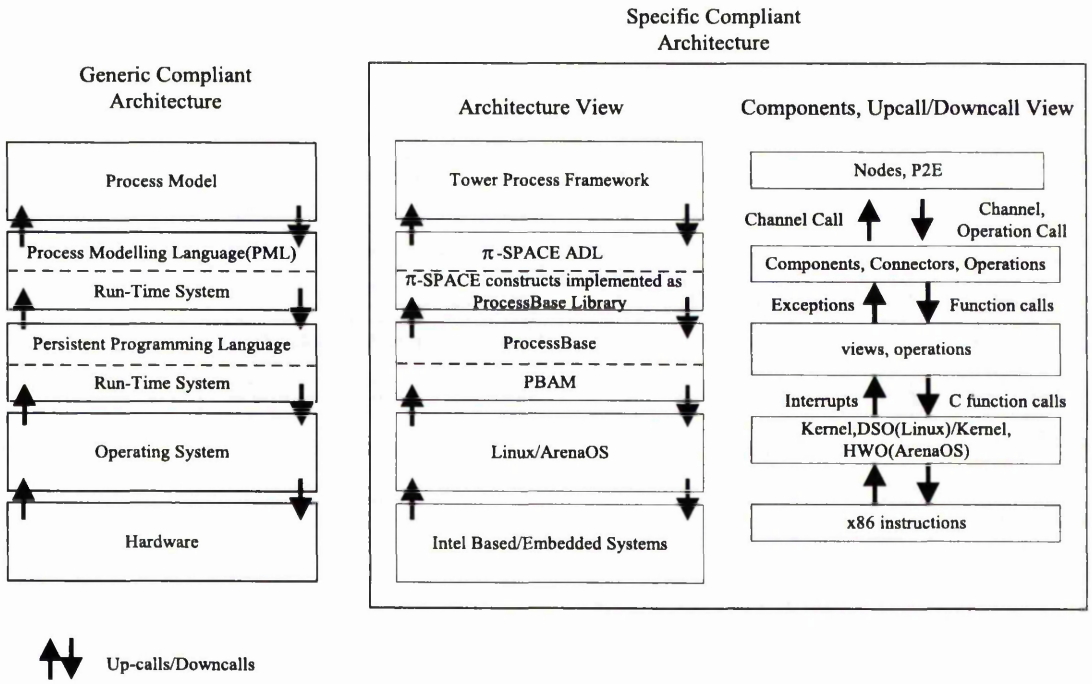

Figure 7.5: A Final Compliant Systems Architecture model

Chapter 8

Discussion and Future Work

8.1 Introduction

This chapter provides a summary of the conclusions that were derived from the results that were collated from the evaluation described in chapter 7. From this set of discussions, some possible avenues of future research are also outlined.

8.2 Compliance Model on the PSEE

8.2.1 Determination of a csa

The csa-model was an attempt to formalise the abstract notion of compliance and make it sufficiently concrete to be applied to any software system. This approach was sufficient for determining compliance but was insufficient for fine-grained measurements of compliance. The degree of compliance should be a good indicator of the amount of effort which is useful in determining the level of flexibility of a systems architecture.

8.2.2 A model of Active Compliance

The previous chapters described an experiment that was designed to demonstrate that a csa-model can be constructed for a PSEE and that it provide better support for the deviated form of evolution. The experiment showed that the csa-model is useful for describing systems in terms of their components at varying levels of abstractions. The work in implementing active compliance, which allows the

support for the deviated form of evolution was completed at a design level, but it is clear that a model can be derived from this in a straightforward manner. This is illustrated in figure 8.1 where the meta-process model is constantly receiving feedback.

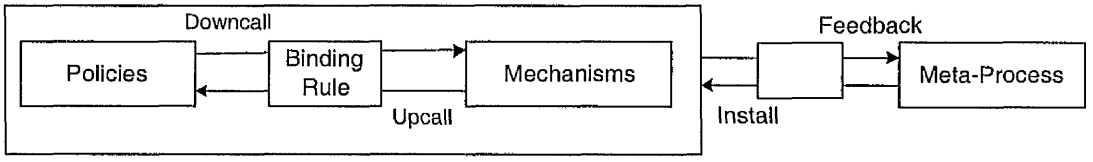

Figure 8.1: The model of Active Compliance

The useful upcalls and downcalls are the feedback and install operations.

8.3 HyperCode and the π-SPACE language

The π-SPACE language was a work in progress during the research and thus constant revisions were made even during the construction of the compiler. This resulted in changes in the syntax and the semantics which changed the code generation rules but such 'deviated' evolution proved to be of benefit in demonstrating the robustness of the csa approach to compiler construction. The final form of the BNF and the code generation rules that forms the derived final enactable π-SPACE are detailed in appendix A.

The main issues during the design and implementation arose because of its specification biased design and much of the effort was spent on removing some language features, which were not relevant to the experiment, and introducing some enactable elements which are useful in our construction of the prototype.

The work done on HyperCode for the π-SPACE language shows that the operations that have been used are sufficiently generic for supporting most programming languages. There were some issues with using π-SPACE though and they are listed as follows:-

1. Specification biased

   The specification biased focus of the language resulted in a few key issues during the initial design and the later stages. During the earlier stages, this resulted in a BNF which is rather huge and complicated. Many attempts were made in order to simplify the language while retaining its main elements which would useful for model checking.

2. Type rules
   Names within the π-SPACE and π-calculus in particular do not really have the notion of types which is required for generating a compiler. Some types were implicitly derived from some examples. Also, the types were not considered as a first class entity within the language.

3. Partial evaluation
   In the π-SPACE language, some of the operations do not return a value immediately as the models have not completed their execution. The end result is that during an Eval operation, the HCA is left waiting for a result to be returned. Semantically this is correct in terms of how the language is defined. All indications are that this could be a user interface problem where some research inputs from the area would help to understand the relationship between the hyperlinks and the operations that the user would be able to perform on the hyperlinks.

8.4 Compliance as a method for construction

Within this project, the csa model was initially used as a model to determine if a software layer is compliant to another layer. Over the course of the project, the csa model, has in fact been used implicitly to guide the constructing of the layers. The awareness of policies, mechanisms and binding rules within each layer resulted in an informal method where compliance determination is used as a form of feedback to the method. As such, the layers integrate without too much effort as the initial effort had already been spent in constructing the system based on the set of policies and mechanisms that were derived during construction.

8.5 The CSA Tools

As the basic CSA tools were used for the experiment in creating a csa-based PSEE, a discussion on their utility is useful. It should be noted that the CSA Tools were created with support for the notion of compliance during the CSA project[57, 58] but there were no example applications that were created in order to experiment their utility prior to this research. The work that is detailed in this thesis should be viewed as the first attempt at defining a concrete definition

and also as the first user of the CSA tools for constructing an application that covers all layers of an application.

Having created the software using the tools, the most fundamental notion of a csa is that of extensibility. The extensibility of a csa goes beyond that of conventional systems due to the provisions for allowing changes to all software layers not only from within the same layer but also from another compliant layer.

In order to describe this clearly, figure 8.2 shows the different approaches for extending a software layer on a conventional system from that of a compliant system.

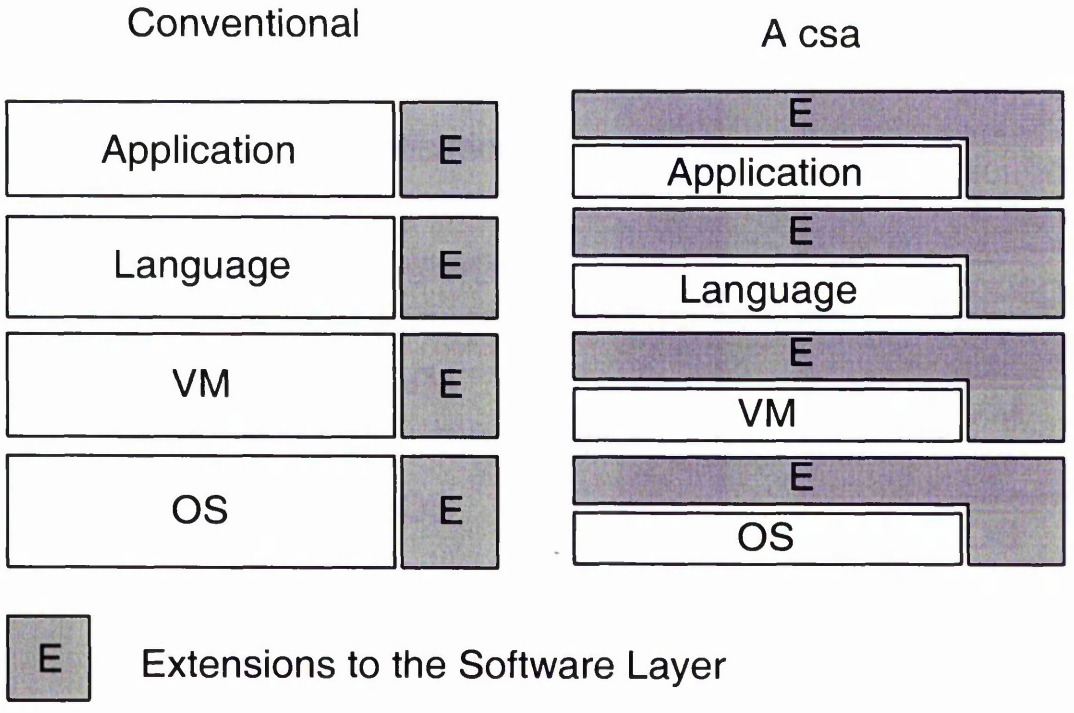

Figure 8.2: The CSA model of extending mechanisms

Conventional systems allow changes that are confined to a given layer and the underlying layer is considered as immutable. In contrast the csa model allows changes which are not confined to its own layer. This is the purpose of including the dynamic loader within the ArenaOS, the opcode invocation to the PBAM from within the ProcessBase language and the HCOs that interact directly with its server within the ProcessBase layer. This model is also reflected in the design of the πPVM where the Operation types within the language are implemented as

ProcessBase functions.

The CSA model thus sees the layers as all being mutable and this view assumes that each are bound to change and that therefore there are mechanisms in place to allow them to change. The only drawback of this approach is that of safety, the general solution being the formalisation of meta-processes to deal with feedback and change.

The original set of HyperCode operations were applied to the π-SPACE language in order to explore how a HyperCode representation might contribute to the expressiveness compared to a text only format. This work has shed some light on how the HyperCode System can be extended to provide a suitable form of representation on executing component-based languages.

8.6 Future Work

During the evaluation of the csa, the Arena OS was utilised as a compliant system that is flexible enough to allow changes to its underlying core. This allowed the mechanisms provided by the operating system to be fine-tuned and customised for the ever-changing policies. This customisation is not unlike the code changing activities of open-source operating systems such as Linux[13] and FreeBSD[49]. However, the compliant nature of the Arena OS by composing it from components of mechanisms and policies, coupled with an explicit binding rule, has set the stage for its next evolution. The implementation of a dynamic loader is detailed in [10]. This is key feature which should allow dynamic compliant support for formalised models. It should also be noted that some of the future work detailed here is to be addressed in the ongoing ArchWare[61] project.

8.6.1 Language Compliance

There were attempts to simplify the language in an architectural framework [29] with hypercode. The work involved the removal of the basic architectural abstractions in order to create a layered language. The basic ArchWare ADL supports a pure π-calculus programming approach. The language has since evolved into a hybrid π-calculus together with an expressions based language. The π-calculus was used to specify the structure of abstractions. The extra language expressions, such as loops and conditional statements, were introduced to bring a more conventional programming element to the language. This is akin to the approach

taken in designing the π-SPACE language in that the π-SPACE was used more as an ADL for structuring the process elements and, the ProcessBase language, in the form of Operations in π-SPACE and Annotations, was used to provide the process programming element.

8.6.2 Compliance in Hardware

The work has only described compliance in software and the mapping of the different compliant properties over different software layers. However, it does not seem that too far fetched a notion for the csa model to be extended to the hardware layer. This has already been attempted in the form of embedded systems, with bespoke embedded processors, that best support the needs of their target application domain. These processors has been built for a specific set of applications and thus the mechanisms are not extensible. However, it is not impossible to imagine the design of a reconfigurable processor. This has in some way been attempted by code morphing processors such as Intel's Itanium and Transmeta's Efficien processors. The promise of nanotech technologies should also allow the processor changes to be performed on the fly. The possiblities of a nano machine that changes its instruction opcodes based on the downcall/upcall from the software will perhaps open up many exciting opportunities.

The challenge will be in building such processors and then mapping the mechanisms and policies of an application onto the downcall and upcall operations between the software and hardware layer. It would thus be interesting to see how the csa-model could be applied to this layer of software to hardware interface.

8.6.3 Mechanisms and Policies as processes

Perry[70] described how tools can be describe in terms of mechanism, policy and structure. Here, we have shown that the systems architecture can also be described in terms of mechanism and policy, and thus open up the possibility that the system is able to manipulate the entire structure by itself.

8.6.4 From Determination to Measurement

The current model only deals with determination of compliance. This level of granularity is rather crude in that we can either tell if something is compliant or not. However, there will be cases where the level of compliance can be measured.

This could result in the notion of partial compliance. This can also be a guide to developing a process for making something compliant, ie how many policies are still not supported, which would then guide the possible approaches required to make a system compliant.

8.6.5 Derived Work

This section described some ongoing work which were derived from the work detailed in this thesis. The ArchWare[61] project is addressing some of the issues addressed in this chapter. The ArchWare ADL[29, 59] simplified the language design by implementing it in layers where each construct was now strongly typed. Even though the original library structures were adopted, they were rewritten for the ArchWare ADL. The approach of compiling the ADL into the ProcessBase language is still used by the ArchWareADL. Simplifying the π-SPACE language also resulted in several languages which focusses on different aspects within the original π-SPACE language. This resulted in for example, the ADL Analysis Language(AAL), ADL Refinement Language(ARL) and ADL Style Language(ASL) to name but a few.

The approach for constructing a hypercode for π-SPACE was also adopted as the model for the hypercode for the ArchWare ADL. A more recent publication which details the use of hypercode in the ArchWare ADL for supporting feedback and change in self-adaptive systems is provided by Balasubramaniam[7].

8.7 Summary

Current PSEEs built using machine-based paradigms have, to date, been inadequate as support tools for software development processes. The resultant static system is unable to cater for the inherently dynamic nature of the supported process models.

The CSA approach does not negate the entire engineering approach but rather augments it by suggesting that application needs should drive the underlying architecture of the system. This is in sharp contrast to the current approach of building the underlying architecture to be as generic as possible in order to cater for all possible applications that will run on top of the architecture.

Each compliant layer of the systems architecture was constructed and tested to be compliant at their own layer. The systems architecture was then created by

integrating all these layers and retested to see if the mechanisms provided by the integrated application continue to be compliant to the policy needs of a process model. Some examples of non-compliance were also outlined where an approach to rectify them was described. This demonstrated a concrete example of how a compliant system are able to provide flexibility by providing the ability to be made compliant.

A sample process model was then specified to illustrate the two forms of evolution in order to illustrate how well a system that has been built with the CSA approach could cope with the evolution requirements of a PSEE. The designed form of evolution has been well supported by current process languages. However, the deviated form of evolution which requires changes to be made which were not anticipated by the original assumptions, is not well supported.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology, Vol 6, No. 3, July 1997, Pages 213-249*, 1997.

[3] G.K. Ananthasuresh and S. Kota. Designing compliant mechanisms. *Mechanical Engineering*, 117(11):93–96, 1985.

[4] Mike Anderson and Phil Griffiths. The nature of the software process modelling problem is evolving. In *Proceedings of the Third European Workshop on Software Process Technology, Villard de Lans, Volume 772 of Lecture Notes in Computer Science*, pages 31–34. Springer-Verlag, February 1994.

[5] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, 1992.

[6] W. R. Ashby. *An introduction to cybernetics.* Chapman, 1956.

[7] D Balasubramaniam, R Morrison, K Mickan, GNC Kirby, BC Warboys, I Robertson, B Snowdon, RM Greenwood, and W Seet. Support for feedback and change in self-adaptive systems. In *ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04), Newport Beach, CA, USA*. ACM, 2004.

[8] K. Beck. *Extreme Programming explained.* Addison-Wesley, 1999.

[9] Stafford Beer. *Designing Freedom.* John Wiley and Sons, 1974.

[10] S Beyer, K Mayes, and B.C Warboys. Application-compliant networking on embedded systems. *5th IEE International Workshop on Networked Appliances*, 2002.

[11] Grady Booch. *Object-Oriented Analysis and Design With Applications.* Addison-Wesley, 1994.

[12] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modelling Language User Guide.* Addison-Wesley, 1998.

[13] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly, 2003.

[14] Frederick P. Jr. Brooks. *The Mythical Man-Month : Essays on Software Engineering.* Addison-Wesley, 1995.

[15] Manfred Broy. Toward a mathematical foundation of software engineering methods. *IEE Transactions on Software Engineering*, 27(1), August 2001.

[16] R.F. Bruynooghe, J.M. Parker, and J.S. Rowles. PSS: A System for Process Enactment. In *Proceedings of the First International Conference on the Software Process: Manufacturing Complex Systems, California, USA*, pages 128–141, October 1991.

[17] Ethan Cerami. *Web Services Essentials.* O'Reilly, 2002.

[18] Christelle Chaudet, R.M. Greenwood, Flavio Oquendo, and Brian Warboys. Architecture-driven software engineering: Specifying, generating and evolving component-based software systems. *IEE Proceedings: Software*, 147:203–214, 2000.

[19] Reidar Conradi, Christer Fernstrom, Alfonso Fuggetta, and Robert Snowdon. Towards a Reference Framework for Process Concepts. *Software Process Technology, 2nd European Workshop, EWSPT 1992, volume 635 in Lecture Notes in Computer Science*, 1992.

[20] Gianpaolo Cugola and Carlo Ghezzi. Software processes: a retrospective and a path to the future. *Sotware Process - Improvement and Practise*, 4, 1998.

[21] B. Curtis, M. I. Kellner, and J. Over. Process Modelling. *Communications of the ACM, 35(9)*, 9 1992.

[22] Darren Dalcher. Feedback, Planning and Control-A Dynamic Relationship. *FEAST 2000 International Workshop on Feedback and Evolution in Software and Business Processes*, 2000.

[23] A.J.T. Davie and R. Morrison. *Recursive Descent Compiling.* Ellis Horwood Limited, 1981.

[24] Tom DeMarco and P.J. Plauger. *Structure Analysis and System Specification.* Prentice Hall, 1985.

[25] Edsger W. Dikjstra. The Structure of the T.H.E Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[26] M. Dowson, B. Nejmeh, and W. Riddle. Fundamental Software Process Concepts. *Proceedings of the first European Workshop on Software Process Modelling*, 1991.

[27] P. H. Feiler and W. S. Humphrey. Software process development and enactment: Concepts and definitions. In *Proceedings of the Second International Conference on the Software Process, Berlin, Germany*, pages 28–40, 1993.

[28] Alfonso Fuggetta. A Classification of CASE Technology. *Computer*, 26(12):25–38, December 1993.

[29] R. Mark Greenwood, Dharini Balasubramaniam, Sorana Cimpan, Graham N.C. Kirby, Kath Mickan, Ron Morrison, Flavio Oquendo, Ian Robertson, Wykeen Seet, Bob Snowdon, Brian C. Warboys, and Evangelos Zirintsis. Process Support for Evolving Active Architectures. In Flavio Oquendo, editor, *Proceedings of the Ninth European Workshop on Software Process Technology, Helsinki, Finland, Volume 2786 of Lecture Notes in Computer Science*, pages 112–127. Springer-Verlag, September 2003.

[30] R. Mark Greenwood, Dharini Balasubramaniam, Graham Kirby, Ken Mayes, Ron Morrison, Aled Sage, Wykeen Seet, and Brian C. Warboys. The Impact of Software-Architecture Compliance on System Evolution. to appear in Madhavji(ed) Software Evolution, by Wiley, December 2004.

[31] R.M. Greenwood, Ian Robertson, and B.C. Warboys. A Support Framework for Dynamic Organizations. In *Proceedings of the Seventh European Workshop on Software Process Technology, Kaprun, Austria, Volume 1780 of Lecture Notes in Computer Science*, pages 6–20, February 2000.

[32] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall International, 1985.

[33] IEEE. *(ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application: Program Interface (API) [C Language].* IEEE, 1996.

[34] D. C. Ince. *An introduction to discrete mathematics,formal system specification and Z.* Oxford University Press, 1992.

[35] C.B. Jones. *Systematic Software Development using VDM.* Prentice Hall, 1990.

[36] Brian Kernighan. *The C Programming Language.* Addison-Wesley, 1994.

[37] Kazuhiro Kosuge and Masayuki Shimizu. Planar parts-mating using structure compliance. In *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2001.

[38] Philippe Kruchten. *The Rational Unified Process.* Addison-Wesley, 1999.

[39] M. M. Lehman. Process Models, Process Programs, Programming Support. In *Proceedings of the Ninth International Conference of Software Engineering*, pages 14–16, 1987.

[40] M. M. Lehman and L.A. Belady. *Program Evolution: Processes of Software Change in APIC Studies in Data Processing No. 27.* Academic Press Inc.(London) Ltd, 1985.

[41] M.M. Lehman, D.E Perry, and W. M. Turski. Why is it so hard to find Feedback Control in Software Processes?(Invited Presentation). In *Proceedings of the 19th Australasian Computer Science Conference*, February 1996.

[42] M.M. Lehman and Juan F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001.

[43] Tim Lindholm and Frank Yellin. *The Java(TM) Language Specification, Second Edition.* Addison-Wesley, 1999.

[44] David C. Luckham, James Vera, and Sigurd Meldal. *Key Concepts in Architecture Definition Language*, pages 23–45. Cambridge University Press, 2000.

[45] Nazim H. Madhavji. The process cycle. *Software Engineering Journal*, 6(5), September 1991.

[46] Matthew Mason. Compliance and force control for computer-controlled manipulators. *IEEE Trans on Systems, Man, and Cybernetics*, 11(6):418–432, 1981.

[47] K. R. Mayes and J. Bridgland. Arena: a run-time operating system for parallel applications. In *Proceedings of 5th EuroMicro Workshop on Parallel and Distributed Processing (PDP'97)*, 1997.

[48] K.R. Mayes. Trends in operating systems towards dynamic user-level policy provision. Technical report, Department of Computer Science, University of Manchester, 1993.

[49] Marshall K. McKusick, Keith Bostic Michael J. Karels, and John S. Quaterman. *The Design and Implementation of the 4.4 BSD Operating System.* Addison Wesley, 1996.

[50] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEE Transactions on Software Engineering*, 26(1), January 2000.

[51] Sun Microsystems. Handling Errors with Java Exceptions. *http://java.sun.com/doc/books/tutorial/essential/exceptions*, 2003.

[52] Sun Microsystems. Java Technologies. *http://java.sun.com/products*, 2003.

[53] Robin Milner. Elements of interaction. *Communications of the ACM*, 36(1):89–89, 1993.

[54] Robin Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, 1999.

[55] R Morrison, D Balasubramaniam, M Greenwood, GNC Kirby, Mayes K, Munro DS, and BC Warboys. Processbase abstract machine manual (version 2.0.6). Technical report, Universities of St Andrews and Manchester, 1999.

[56] R Morrison, D Balasubramaniam, M Greenwood, GNC Kirby, K Mayes, DS Munro, and BC Warboys. Processbase reference manual (version 1.0.6). Technical report, Universities of St Andrews and Manchester, 1999.

[57] R. Morrison, D. Balasubramaniam, R.M. Greenwood, G.N.C Kirby, K. Mayes, D.S Munro, and B. Warboys. An approach to compliance in software architectures. *IEE Computing and Control Engineering Journal, Special Issue on Informatics*, 11(4):195–200, 2000.

[58] R Morrison, D Balasubramaniam, RM Greenwood, GNC Kirby, K Mayes, DS Munro, and BC Warboys. A compliant persistent architecture. *Software - Practice and Experience, Special Issue on Persistent Object Systems*, 30(4):363–386, 2000.

[59] Ron Morrison, Graham Kirby, Dharini Subramaniam, Flavio Oquendo, Sorana Cimpan, Brian Warboys, Bob Snowdon, and Mark Greenwood. Support for Evolving Software Architectures in the ArchWare ADL. In *Fourth Working IEEE/IFIP Conference on Software Architecture(WICSA'04)*, pages 69–78. IEEE Computer Society, June 2004.

[60] P. Naur and B. Randell. Software engineering: Report on a conference sponsored by the nato science committee. 1969.

[61] Flavio Oquendo, Brian Warboys, Ron Morrison, Regis Dindeleux, Ferdinando Gallo, Hubert Garavel, and Carmen Occhipinti. ArchWare: Architecting Evolvable Software. In *Proceedings of the First European Worshop, EWSA*, pages 257–271, 2004.

[62] L. J. Osterweil. Software Processes are Software Too. In *Proceedings of the Ninth International Conference of Software Engineering*, pages 2–13, 1987.

[63] L. J. Osterweil. Software Process are software too, revisited: an invited talk on the most influential paper of ICSE 9. *Proceedings of the 1997 International Conference on Software Engineering, pages 540-548*, 1997.

[64] Martyn A. Ould. *Business Processes.* John Wiley and Sons, 1995.

[65] D. L. Parnas. On the Criteria to be Used in Decomposing a System into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[66] D. L. Parnas. On a "Buzzword": Hierarchical Structure. *IFIP Congress*, 1974.

[67] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering SE-2,1 (1976): pages 1-9*, 1976.

[68] David L. Parnas. Predicate Logic for Software Engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, September 1993.

[69] David L. Parnas and D.L. Siewiorek. Use of the Concept of Transparency in the Design of Hierachically Structured Systems. *Communications of the ACM*, 18(7):401–408, September 1975.

[70] D.E. Perry and G.E. Kaiser. Models of software development environments. *IEE Transactions on Software Engineering*, 17(3), March 1991.

[71] B. C. Pierce and D.N. Turner. Pict: a programming language based on the pi-calculus. *Indiana University CSCI Technical Report 476, March 1998*, 1998.

[72] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Tickey, and Phil Winterbottom. Plan 9 from bell labs. *Computer Systems*, 8(3), 1995.

[73] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM, 17(7)*, 7 1974.

[74] Dennis M. Ritchie. The Limbo Programming Language. *http://www.vitanuova.com/inferno/papers/limbo.html*, 2000.

[75] Ian Robertson. An implementable meta-process. In *Proceedings of Second World Conference on Integrated Design and Process Technology*, 1996.

[76] Ian Robertson. An evolutionary approach to process engineering. In *Proceedings of 1st International Workshop on the Many Facets of Process Engineering*, pages 159–163, 1997.

[77] Stuart Sechrest. An introductory 4.4bsd interprocess communication tutorial. Technical report, Computer Science Research Group, Computer Science Dvivion, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1993.

[78] Wykeen Seet. An evaluation of a generic software process framework. MPhil Thesis, The University of Manchester, 2000.

[79] Wykeen Seet, Ian Robertson, and Brian Warboys. Enactable evolution in the software development process. *FEAST 2000 International Workshop on Feedback and Evolution in Software and Business Processes*, 2000.

[80] Wykeen Seet and Brian Warboys. A Compliant Environment for Enacting Evolvable Process Models. In *Proceedings of the Ninth European Workshop on Software Process Technology, Helsinki, Finland, Volume 2786 of Lecture Notes in Computer Science*, pages 154–163. Springer-Verlag, September 2003.

[81] Mary Shaw and Paul Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings of the 21st International Computer Software and Applications Conference*, pages 6–13, August 1997.

[82] Mary Shaw and David Garlan. *Formulations and Formalisms in Software Architecture*, volume 1000 of *Lecture Notes in Computer Science*, pages 307–323. Springer-Verlag, 1995.

[83] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an emerging discipline*. Prentice Hall, 1996.

[84] R. A. Snowdon. Active Models and Process Support. In *Proceedings of the*

Fifth European Workshop on Software Process Technology, Kaprun, Austria, Volume 1149 of Lecture Notes in Computer Science, pages 93–98, October 1996.

[85] R.A. Snowdon. An Introduction to the IPSE 2.5 Project. *ICL Technical Journal 6(3)@ 467-478*, 1989.

[86] Ian Sommerville and Simon Monk. Supporting informality in the software process. In *Proceedings of the Third European Workshop on Software Process Technology, Villard de Lans, France, Volume 772 of Lecture Notes in Computer Science*. Springer-Verlag, February 1994.

[87] Ian Sommerville and Tom Rodden. Understanding the software process as a social process. In *Proceedings of the Second European Workshop on Software Process Technology, Trondheim, Norway, Volume 635 of Lecture Notes in Computer Science*. Springer-Verlag, September 1992.

[88] Richard Stevens. *Unix Network Programming*. Prentice Hall, 1998.

[89] S. M. Jr. Sutton, D. Heimbigner, and L. J. Osterweil. Language Constructs for Managing Change in Process-Centred Environments. In *Proceedings of the Fourth ACM SIGSOFT/SIGPLAN Symposium: Practical Software Development Environments*, pages 206–207. ACM Press, 1990.

[90] S. M. Jr. Sutton, B. S. Lerner, and L. J. Osterweil. Experience Using the JIL Process Programming Language to Specify Design Processes. Technical Report UM-CS-1997-068, Computer Science Department, University of Massachusetts, , 1997.

[91] S. M. Jr. Sutton and L. J. Osterweil. The Design of a Next-Generation Process Language. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97, Volume 1301 of Lecture Notes in Computer Science)*, pages 142–158. Springer-Verlag, 1997.

[92] Clemens Szyperski. *Component Software and the Way Ahead*, pages 1–20. Cambridge University Press, 2000.

[93] B. C. Warboys, D. Balasubramaniam, R.M. Greenwood, G. N. C. Kirby, K. Mayes, R. Morrison, and D. Munro. Instances and connectors: Issues for

a second generation process language. In *Proceedings of the sixth European Workshop on Software Process Technology, Weybridge, UK, Volume 1487 of Lecture Notes in Computer Science*, pages 137–142. Springer-Verlag, 1998.

[94] B.C. Warboys. *The IPSE 2.5 Project: A Process Model Based Architecture*, pages 313–331. Ellis Horwood Limited, 1989.

[95] B.C. Warboys. The IPSE 2.5 Project: Process Modelling as the Basis for a Support Environment. *Proceedings of the First International Conference on System Development Environments and Factories, pages 59-74*, 1989.

[96] BC Warboys, D Balasubramaniam, RM Greenwood, GNC Kirby, K Mayes, R Morrison, and DS Munro. Collaboration and composition: Issues for a second generation process language. In O. Nierstrasz and M. Lemoine, editors, *Proceedings of the Seventh European Software Engineering Conference, Toulouse, France, Volume 1687 of Lecture Notes in Computer Science*, pages 75–91. Springer-Verlag, September 1999.

[97] Brian Warboys. The Practical Application of Process Modelling - some early reflections. In *Proceedings of the first European Workshop on Software Process Modelling, Milan 1991*, 1991.

[98] Brian Warboys. The software paradigm. *ICL Technical Journal*, 10(1), May 1995.

[99] Brian Warboys, Peter Kawalek, Ian Robertson, and Mark Greenwood. *Business Information Systems: A Process Approach.* McGraw-Hill International (UK) Limited, 1999.

[100] Anthony I. Wasserman. Toward a discipline of software engineering. *IEE Software*, 13(6):23–31, November 1996.

[101] Lancelot Law Whyte. Structural Hierarchies: A Challenging Class of Physical and Biological Problems. In *Hierarchical Structures*, pages 3–16, November 1968.

[102] J. B. Wordworth. *Software Engineering with B.* Addison-Wesley, 1996.

[103] Benjamin Yeomans. Enhancing the world-wide-web, third year project report. Technical report, Department of Computer Science, University of Manchester, 1996.

[104] Edward Yourdon. *Modern Structured Analysis.* Yourdon Press Computing Series. Prentice-Hall International, 1989.

[105] E Zirintsis. *Towards Simplification of the Software Development Process: The Hyper-Code Abstraction.* Phd, University of St Andrews, 2000.

[106] E Zirintsis, GNC Kirby, and R Morrison. Hyper-code revisited: Unifying program source, executable and data. In *9th International Workshop on Persistent Object Systems, Lillehammer, Norway*, 2000.

Appendix A

Enactable π-SPACE

A.1 Introduction

The Enactable π-SPACE language is a derivation of the the original π-SPACE. Enactable π-SPACE is designed to be enactable on the πPVM. The following sections details the Reserved Words, The Grammar in BNF and Code Generation Rules of the Enactable π-SPACE language.

A.2 Reserved Words

The following are the reserve words of the Enactable π-SPACE language.

`attach`	`behaviour`	`component`	`connector`	`define`
`decompose`	`in`	`inout`	`recompose`	`replace`
`new`	`operation`	`out`	`port`	`type`
`whenever`	`where`			

A. 3 Grammar in EBNF

Format

1. terminals – terminals
2. <non-terminals> - non-terminals
3. code_gen – Basic code generation operations

1. π-space program

1.1 *<π-space architecture>* ::=	[<list of declarations>] <architecture>
1.2 *<list of declarations>* ::=	<type declaration>[<annotation>][; <type declaration>[<annotation>]]*
1.3 *<type declaration>* ::=	define <declaration>
1.4 *<declaration>* ::=	<port type declaration> | <operation type declaration> | <behaviour type declaration>| <component type declaration>| <connector type declaration>| <composite type declaration>

2. Port Type declaration

2.1 *<port type declaration>*::=	port type <port type name> [[<list of typed parameters>]] { [<list of typed_parameter> ,]<port specification> }
2.2 *<list of typed parameters>*::=	<typed parameter>[, <typed parameter>]*
2.3 *<typed parameter>*::=	<parameter name>:<parameter type name>
2.4 *<port specification>*::=	<port signature> = <port protocol>
2.5 *<port signature>*::=	<port type name> [[<list of parameter names>]]
2.6 *<list of parameter names>*::=	<parameter name> [, <parameter name>] *
2.7 *<port protocol>*::=	<port actions> [<pi-calculus operator> <port actions>]*
2.8 *<port actions>*::=	<send channel> | <receive channel> | <port signature> | <primitive process>
2.9 *<send channel>*::=	<channel name>_< [<list of parameter names>] >
2.10 *<receive channel>*::=	<channel name> (([<list of parameter names>]))

2.11 <*pi-calculus operator*>::= ∘ | + | | | !

2.12 <*primitive process*> ::= $

3. Operation Type declaration

3.1 <*operation type declaration*> ::= operation type <operation type name>
[[<list of operation type parameter>]]
{ <ProcessBase code> }

3.2 <*list of operation type parameter*> ::= <operation type parameter> [,
<operation type parameter>]*

3.3 <*operation type parameter*> ::= in [<list of typed parameter>]
| out [<list of typed parameter>]
| inout [<list of typed parameter>]

4. Behaviour Type declaration

4.1 <*components behaviour type declaration*>::=
behaviour component type <component
behaviour type name>
[<list of typed ports>]
[<variable_declarations>,]*
{ <component_behaviour_specification> }

4.2 <*connectors behaviour type declaration*>::=
behaviour connector type <connector behaviour
type name>
[<list of typed ports>]
{
[<list_of_typed_parameter>,]
<connectors behaviour specification>
}

4.3 <*list of typed ports*>::= <port name> : <description of typed port>
[,<port name> : <description of typed port>]*

4.4 <*description of typed port*> ::= <port type name> [<list of typed parameters>]

4.5 <*variable_declarations*> ::= <typed_parameter> | <operation_decl>

4.5 <*operation_behaviour_decl*> ::= <operation_name>[[<list of operation type
parameter>]]
{ <ProcessBase code> }

4.6 <*components behaviour specification*>::=
<behaviour signature> =
<component behaviour protocol>

4.7 <*connectors behaviour specification*>::=
<behaviour signature> =
<connector behaviour protocol>

4.8 <*behaviour signature*>::= <behaviour type name> [[<list of port names>]]

4.9 <*list of port names*>::= <port name> [, <port names>]*

4.10 <*component behaviour protocol*>::= <component behaviour actions>
[<pi-calculus operator>
<component behaviour actions>]*

4.11 <*connector behaviour protocol*>::= <connector behaviour actions>
[<pi-calculus operator>
<connector behaviour actions>]*

4.12 <*component behaviour actions*>::= <operation >
| <behaviour signature>
| <communication>
| <primitive process>

4.13 <*connector behaviour actions*>::= <behaviour signature>
| <communication>
| <primitive process>

4.14 <*operation*>::= <operation name>
[[[<list of parameter names>]]]

4.15 <*communication*>::= <port name>@<channel>

4.16 <*channel*>::= <send channel> | <receive channel>

5. Component Type Declaration

5.1 <*component type declaration*>::= component type <component type name> [[<list of typed ports>]]
{ <port_behaviour_decl> [||
<port_behaviour_decl>]* }

5.2 <*port_behaviour_declaration*>::= <port declaration> | <behaviour declaration>

5.3 <*port declaration*>::= port <port name> : <port type name>
[[<list of parameter names>]]

5.4 <*behaviour declaration*>::= behaviour <behaviour name> : <component behaviour type name>
[[<list_of_typed_parameters>]]

6. Connector Type Declaration

6.1 <*connector type declaration*>::= connector type <connector type name>
[<list of typed ports >]
{<port_behaviour_decl> (||
<port_behaviour_decl>)* }

7. Composition Type Declaration (Includes the Evolution composition)

7.1 <*architecture*> ::= <composite>

7.2 <*composite type declaration*> ::= composite type <composite type name>
[[<list of typed architectural elements>]]
<core declaration>

7.3 <*list of typed architectural elements*> ::= <typed_parameter >
[, <typed_parameter >]*

7.4 <*composite*> ::= compose <composite name> <core declaration>

7.5 <*core declaration*> ::= {
[<model operation declarations>]
[where <where declarations>]
[whenever <whenever declarations>]
}

7.6 <*model operation declarations*> ::=<model operation declaration>
(|| <model operation declaration>)

7.7 <*model operation declaration*> ::= <decompose operation> |
<architecture element declaration>

7.8 <*decompose operation*>::= decompose <composite name>

7.9 <*architectural element declaration*>::= <typed_parameter>
[[[<list of renaming ports>]]]
[|| <typed_parameter>
[[[<list of renaming ports>]]]
]*

7.10 <*list of renaming ports*> ::= <renaming port> [, <renaming port>]*

7.11 <*renaming port*> ::= <renaming port name>
[{ <list of renaming channels> }]

7.12 <*renaming port name*> ::= <port name> [/ <new port name>]

7.13 <*list of renaming channels*> ::= <renaming channel> [, <renaming channel>]*

7.14 <*renaming channel*> ::= <channel name> / <new channel name>

7.15 <*where declarations*> ::= <where declaration> [, where declaration]*

7.16 <*where declaration*>::= <replace declaration>
| <attachment declaration>
| <recompose declaration>
| <composite declaration>
| <new decl>

7.17 <*composite declarations*> ::= <composite type name>
[[<architectural elements>]]
[, <composite type name > [[
<architectural elements>]]*

7.18 < *architectural elements*>::= <architectural element>
[, < architectural element>]*

7.19 < *architectural element*>::= <component name> | <connector name>]

7.20 <*replace declaration*> ::= replace <component name> by
<new component name>

7.21 <*attach declaration*> ::= attach <component channel> to
<connector channel>

7.22 <*component channel*> ::= <component name>@<port name>
[@<channel name>]

7.23 <*connector channel*> ::= < connector name>@<port name>
[@<channel name>]

7.24 <*recompose declaration*> ::= recompose (<list of component names>)

7.25 <*list of component names*> ::= <component name> [, <component name>]*

7.26 <*list of operation declarations*> ::= <branch declarations>
[; <branch declarations>]*

7.27 <*branch declarations*> ::= <new declarations> [, <new declarations>]

7.28 <*new declarations*>::= new <new instance> => <new instance operation>

7.29 <*new instance*> ::= <component instance name> | <component channel name>

7.30 <*new instance operation*>::= <new operation>| <attach operation>

7.31 <*new operation*> ::= new <new instance>

8. Annotation

8.1 <*annotation*> ::=	<%ps<value declaration> [; <value declaration>]*%ps>
8.2 <*value_declaration*> ::=	let <identifier> <- <ann_expression>
8.4 <*ann_expression*> ::=	<ann_expression> <add_op> <ann_expression> | <ann_expression> <mult_op> <ann_expression> | <object constructor>| <ann_expression> <deref_op> <ann_identifier>| <ann_literal>
8.5 <*object_constructor*> ::=	<identifier>([<parameter assignment>])
8.6 <*parameter assignment*> ::=	<identifier> <- <ann_expression>[,<identifier> <- <ann_expression>]*
8.7 <*add_op*> ::=	+ | -
8.8 <*mult_op*> ::=	<int_mult_op> | <string_mult_op>
8.9 <*int_mult_op*> ::=	* | div
8.10 <*string_mult_op*> ::=	++
8.11 <*ann_literal*> ::=	<string_literal>|<int_literal>
8.12 <*string_literal*> ::=	« [char]*«
8.13 <*int_literal*> ::=	digit[digit]*
8.14 <*char*> ::=	any ASCII character
8.15 <*digit*> ::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A. 4 Code Generation Rules

Format

1. *identifier* – π-SPACE syntactic constructs that are relevant for code generation
2. π-SPACE - π-SPACE syntactic construct that are not relevant for code generation
3. ProcessBase – Generated ProcessBase text code
4. code_gen – Basic code generation operations

Code Generation Operations

1. type_string(*id*) – returns the type name of the *id*
2. for each id_i do expression end for – loops through the set of *id* and uses the specific *id* indexed by *i* in the expression
3. if *boolean* then $expression_1$ else $expression_2$ – if *boolean* is true then $expression_1$ is executed else *expression*

1. π-space program

π-SPACE	ProcessBase
list of declarations *architecture*	include safeOpLib ioLib ps_utilities ps_com *list_of_declarations* *architecture*

2. Port Type declaration

π-SPACE	ProcessBase

define port type *name*
[*channel_id*$_1$:*c_type*$_1$,..,
channel_id$_n$:*c_type*$_n$]
{
port specification
}
	type *name* is view[typeTag:int;
	channel_id$_1$: *c_type*$_1$;.. ;*channel_id*$_n$:*c_type*$_n$;
	port_spec: string]
	
	! *name* Generator
	let gen_*name*_Port <- fun(
	channel_id$_1$: *c_type*$_1$;.. ;*channel_id*$_n$:*c_type*$_n$)-> *name*
	{
	view(typeTag <- portTag,
	channel_id$_1$ <- *c_type*$_1$,.. ,*channel_id*$_n$<-*c_type*$_n$,! names are maintained
	port_spec <- *port specification*)
	}

3. Operation Type declaration

π-SPACE	ProcessBase
define operation type *name* [in[*id*$_{i1}$: *t*$_{i1}$,..,*id*$_{in}$: *t*$_{in}$], out[*id2*$_1$: *t2*$_1$,..,*id*$_{o1}$:*t2*$_n$], inout[*id3*$_1$: *t3*$_1$,..,*id3*$_n$:*t3*$_n$]] { *ProcessBase_code* } ! in[..], out[…], intout[…] might not appear in ! order and are optional	type *name* is view[typeTag:int; *id*$_1$: *t*$_1$;.. ;*id*$_n$: *t*$_n$; *id2*$_1$:loc[*t2*$_1$] ;.. ;*id2*$_n$:loc[*t2*$_n$] ; *id3*$_1$: loc[*t3*$_1$] ;..;*id3*$_n$: loc[*t3*$_n$]; operation_fun: fun()] ! Code generation for *name* let gen_*name*_Operation<- fun(*id*$_{i1}$: *t*$_{i1}$;.. ;*id*$_n$: *t*$_n$; *id2*$_1$:loc[*t2*$_1$] ;.. ;*id2*$_n$:loc[*t2*$_n$] ; *id3*$_1$: loc[*t3*$_1$] ;..;*id3*$_n$: loc[*t3*$_n$]

	) -> *name*
	{
	let *name*_operation_fun <- fun()
	{
	ProcessBase_code
	}
	view(typeTag<-operationTag;
	id_1 <- id_1 ,.. ,id_n<- id_n,
	$id2_1$<-$id2_1$,.. ,$id2_n$<- $id2_n$,
	$id3_1$<- $id3_1$,..,$id3_n$<-$id3_n$,
	operation_fun <- *name*_operation_fun
	)
	}

4. Behaviour Type declaration

4.1 behaviour type declaration and generator

π-SPACE	ProcessBase

behaviour component type *name*
[$port\_id_1$: p_type_1,..,$port\_id_n$:p_type_n]
{
id_1 : t_1,..,id_n: t_n, ! id declarations
op_1[p_1]{$pbase_1$},..,op_n[p_n]{$pbase_n$} !op decls
 	
name[$port\_id_1$,..,$port\_id_n$]=
components behaviour specification
}
	type *name* is view
	[typeTag :int ;
	$port\_id_1$: p_type_1;.. ;$port\_id_n$:p_type_n;
	behaviour_fun : loc[fun()] ;
	behaviour_spec : string]
	
	! Instance Generator
	let gen_*name*_Behaviour <-fun(
	$port\_id_1$: p_type_1;.. ;$port\_id_n$:p_type_n
	) -> *name*
	{
	! Generate optional_operations
	for each op_i
	let op_i <- fun(p_i) ! parameters p
	{ $pbase_i$ }
	end for
	
	! Generate behaviour_functions
	let *name*_behaviour_fun<- fun()
	{
	for each id_i
	let id_i <- loc(if type_string(t_i) ==string « « else 0)
	end if
	component_behaviour_specification
	}
	
	view(
	typeTag <- behaviourTag,
	$port\_id_1$ <- $port\_id_1$,
	…,
	$port\_id_n$ <- $port\_id_n$,
	behaviour_fun <- loc(*name*_behaviour_fun),
	behaviour_spec <- *behaviour_spec*
	)
	}

4.2 behaviour pi-calculus code generation

4.2.1 sequence(.)pi_operator

π-SPACE	**ProcessBase**
$pi\_expression_1$. $pi\_expression_2$....$pi\_expression_{nn}$	$pi\_expression_1$ $pi\_expression_2$ .. $pi\_expression_n$

4.2.2 conditional (+) pi_operator

π-SPACE	**ProcessBase**
$Pi\_expression_1$+$pi\_expression_2$...+$pi\_expression_n$	if checkReceiveChannel(first_exp($pi\_expression_1$)) then $pi\_expression_1$ else if checkReceiveChannel(first_exp($pi\_expression_2$)) then $pi\_expression_2$ ... else if checkReceiveChannel(first_exp($pi\_expression_n$)) then $pi\_expression_n$ else { ! do nothing } ! where first_exp, gets the first channel exp in the ! pi_expression, codegen assumes that condition is ! based on channels, but can be extended

4.2.3 parallel (||) pi_operator

π-SPACE	**ProcessBase**
$Pi\_expression_1$||$pi\_expression_2$...||$pi\_expression_n$	let thread_list <-newThread(fun(){$pi\_expression_1$}, nil(ThreadList) thread_list :=newThread(fun(){$pi\_expression_2$},thread_list} ... thread_list :=newThread(fun(){$pi\_expression_n$},thread_list} wait_thread_termination(thread_list)

4.3 pi_expressions code generation

4.3.1 send_channel expression

π-SPACE	**ProcessBase**
channel<identifier>	if type_string(channel) = channel_string sendStringTo(*channel, 'identifier*) else if type_string(channel) = channel_int sendIntTo(*channel, 'identifier*)

4.3.2 receive_channel expression

π-SPACE	ProcessBase
channel(*identifier*)	if type_string(channel) = channel_string receiveStringFrom(*channel,identifier*) else if type_string(channel) = channel_int receiveIntFrom(*channel, identifier*)

4.3.2 operation expression

π-SPACE	ProcessBase
operation[*param*$_1$, …, *param*$_n$]	*operation*(*param*$_1$, …, *param*$_n$)

5. Component Type Declaration

π-SPACE	ProcessBase

define component type *name*
[*port_id*$_1$: *p_type*$_1$,..,*port_id*$_n$:*p_type*$_n$]
{
port *port_id*$_1$: *p_type*$_1$,.., ||
port …||
port *port_id*$_n$:*p_type*$_n$ ||
behaviour *behaviour_id*$_1$: *b_type*$_1$ ||
behaviour … ||
behaviour *behaviour_id*$_n$: *b_type*$_n$
}
	type *name* is view[
	typeTag : int ;
	port_id$_1$: *p_ type*$_1$;.. ;*port_id*$_n$:*p_ type*$_n$;
	behaviour_id$_1$: *b_ type*$_1$;.. ;*behaviour_id*$_n$:*b_ type*$_n$;
	start_behaviour : loc[fun()]
	]
	
	! Instance Generator
	let gen_*name*_Component <= fun(
	port_id$_1$: *p_ type*$_1$;.. ;*port_id*$_n$:*p_ type*$_n$;
	behaviour_id$_1$: *b_ type*$_1$;.. ;*behaviour_id*$_n$:*b_ type*$_n$
	) -> *name*
	{
	let *name*_start_behaviour <= fun()
	{
	'(*behaviour_id*$_1$.behaviour_fun)()
	….behaviour_fun)()
	'(*behaviour_id*$_n$. behaviour_fun)()
	}
	
	view(
	typeTag <= componentTag,
	port_id$_1$ <= *port_id*$_1$,.. ,*port_id*$_n$ <= *port_ id*$_n$;
	behaviour_id$_1$ <= *behaviour_ id*$_1$,.. ,
	behaviour_id$_n$ <= *behaviour_ id*$_n$,
	start_behaviour <= loc(*name*_start_behaviour)
	)
	}

6. Connector Type Declaration

π-SPACE	**ProcessBase**

define connector type *name*
[*port_id*$_1$: *p_type*$_1$,..,*port_id*$_n$:*p_type*$_n$]
{
port *port_id*$_1$: *p_type*$_1$,.., ||
port ...||
port *port_id*$_n$:*p_type*$_n$ ||
behaviour *behaviour_name*$_1$: *behaviour_type*$_1$
 	
behaviour ... ||
behaviour *behaviour_name*$_n$: *behaviour_type*$_n$
}
	type *name* is view
	[typeTag : int ;
	port_id$_1$: *p_ type*$_1$;.. ;*port_id*$_n$:*p_ type*$_n$;
	behaviour_id$_1$: *b_ type*$_1$;.. ;*behaviour_id*$_n$:*b_ type*$_n$;
	start_behaviour : loc[fun()]
	]
	
	! Instance Generator
	let gen_*name*_Component<- fun(
	port_id$_1$: *p_ type*$_1$;.. ;*port_id*$_n$:*p_ type*$_n$;
	behaviour_id$_1$: *b_ type*$_1$;.. ;*behaviour_id*$_n$:*b_ type*$_n$;
	start_behaviour : loc[fun()]
	) -> *name*
	
	let *name*_start_behaviour <- fun()
	{
	'(*behaviour_id*$_1$.behaviour_fun)()
	....behaviour_fun)()
	'(*behaviour_id*$_n$: behaviour_fun)()
	}
	
	view(
	typeTag <- connectorTag,
	port_id$_1$ <- *port_id*$_1$,.. ,*port_id*$_n$ <- *port_ id*$_n$,
	behaviour_id$_1$ <- *behaviour_ id*$_1$,.. ,
	behaviour_id$_n$ <- *behaviour_ id*$_n$,
	start_behaviour <- loc(*name*_start_behaviour)
	)
	}

7. Composition Type Declaration

7.1 Composite type declaration and generator

π-SPACE	ProcessBase

define composite type *composite_name*
[$id_1 : t_1$,...,$id_n : t_n$]
compose *name*
{
decompose *composite*
[where *where declarations*]
[whenever *whenever declarations*]
}}
	Type *name* is view
	[typeTag :int ;
	$id_1 : t_1$;
	...;
	$id_n : t_n$;
	where_fun : loc[fun()] ;
	whenever_fun : loc[fun()] ;
	start_behaviour : loc[fun()]
	]
	
	! Instance Generator
	let gen_*name*_Composite<- fun(
	$id_1 : t_1$;
	...;
	$id_n : t_n$
	) -> *name*
	{
	! where declarations
	let *name*_where_fun <- fun()
	{
	where declarations
	}
	
	where_fun() ! executes the where function
	
	! whenever declarations
	let *name*_whenever_fun <- fun()
	{
	whenever declarations
	}
	
	let *name*_start_behaviour <- fun()
	{
	let thread_list <-
	newThread(fun(){'(id_1.start_behaviour)()}, nil(ThreadList)
	thread_list :=newThread(fun(){'(id_2.start_behaviour)(),
	thread_list}
	...
	thread_list :=newThread(fun(){'(id_n.start_behaviour)()},
	,.thread_list}
	wait_thread_termination(thread_list)
	}
	
	view(
	typeTag <- compositeTag,
	id_1 <- id_1;
	...;
	id_n <- id_n;
	where_fun <- loc(*name*_where_fun) ;
	whenever_fun <- loc(*name*_where_fun) ;
	start_behaviour <- loc(*name*_start_behaviour)
	)
	}

7.2 Architecture Compose operation

π-SPACE	**ProcessBase**

compose *name*
{ $id_1 : t_1$‖
...‖$id_n : t_n$
decompose *composite*
[where *where declarations*]
[whenever *whenever declarations*]
}
	let *name*<- fun(
	$id_1 : t_1$;
	…;
	$id_n : t_n$
	) -> view[typeTag :int ;
	$id_1 : t_1$;
	…;
	$id_n : t_n$;
	wherefun : loc[fun()] ;
	whenever_fun : loc[fun()] ;
	start_behaviour : loc[fun()]
	]
	{
	! where declarations
	let *name*_where_fun <- fun()
	{
	where declarations
	}
	*name*_where_fun() ! executes the where function
	
	! whenever declarations
	! whenever declarations
	let *name*_whenever_fun <- fun()
	{
	whenever declarations
	}
	
	let *name*_start_behaviour <- fun()
	{
	let thread_list <-
	ewThread(fun(){'(id_1.start_behaviour)()}, nil(ThreadList)
	thread_list :=newThread(fun(){'(id_2.start_behaviour)(),
	thread_list}
	…
	thread_list :=newThread(fun(){'(id_n.start_behaviour)()},
	,.thread_list}
	wait_thread_termination(thread_list)
	}
	
	! Start the model
	name.start_behaviour()
	view(
	typeTag <- compositeTag,
	id_1 <- id_1;
	…;
	id_n <- id_n;
	where_fun <- loc(*name*_where_fun) ;
	whenever_fun <- loc(*name*_whenever_fun) ;
	start_behaviour <- loc(*name*_start_behaviour)
	)
	}

7.3 where declarations

π-SPACE	ProcessBase
attach *channel_a* to *channel_b* replace *component_a* by *component_c* recompose (*component*$_1$, *component*$_2$, ...,*component*$_n$)	attachChannel(any(*channel_a*) , any(*channel_b*)) replaceComponent(any(*component_a*), any(*component_b*)) let componentList <= addComponent(any(*component*$_1$) ,componentList) componentList := addComponent(any(*component*$_2$) ,componentList) ... componentList := addComponent(any(*component*$_n$) ,componentList) reconposeComposite(componentList)

7.4 whenever declarations

π-SPACE	ProcessBase
whenever *whenever_declarations* new *component* => new *new_component* new *component* => attach *channel_a* to *channel_b*	! Still working on possible code generation strategies

8. Annotation

8.1 annotation expression body

π-SPACE	ProcessBase
<%ps let *id*$_1$ <= *ann_expression*$_1$ ... let *id*$_n$ <= *ann_expression*$_n$ %ps>	let *id*$_1$ <= *ann_expression*$_1$ ... let *id*$_n$ <= *ann_expression*$_n$! No changes except the different ann_expression specified ! later

8.2 annotation expression

π-SPACE	ProcessBase
ann_expression	ann_ expression ! No conversion, follow PBase expressions

8.3 annotation component instance construction

π-SPACE	ProcessBase
identifier(*param*$_1$<- *d*$_1$, ..,*param*$_n$<= *id*$_n$)	gen_*identifier*_type_string(*identifier*)(*id*$_1$, ..., *id*$_n$)

8.4 channel type instance construction

π-SPACE	**ProcessBase**

[*identifier*]	{
	let x <- any(*identifier*)
	project x into X
	string : gen_channel_string(*identifier*);
	int : gen_channel_int(*identifier*);
	default: gen_channel_string("")
	}

9. Channel Type Declaration

π-SPACE	**ProcessBase**
[*type_id*]	if type_string(*type_id*) = « string » channel_string else if type_string(type_id) = « int » channel_int

Appendix B

The Tower Model

The basic Towers framework was specified in the π-SPACE language in order to verify its feasibility. This work was done mainly in collaboration with the Informatics Process Group at the University of Manchester where the model has an enactable model on Process *Web*. This section shows the core classes that are relevant to construct a Tower Node.

B.1 Towers in π-SPACE

B.2 HDev Node Component

The HDev node is just a basic shell that will be bound with specific methods. The core components of *Specification* and *Product* are included in this component.

```
!COMPONENTS

!NODE COMPONENT
define component type HdevNode[node-parent:
      BiDiPort(inchan:[Specification], outchan:[Specification]],
 node-child:  BiDiPort(inchan:[Specification],
      outchan:[Specification]],
 node-specMethod:  BiDiPort(inchan:[Specification],
      outchan:[Specification]],
 node-verifyMethod:  BiDiPort(inchan:[Specification],
      outchan:[Specification]]
]
{
```

```
  port node-parent : BiDiPort[inchan, outchan]||
  port node-child  : BiDiPort[inchan, outchan]||
  port node-specMethod : BiDiPort[inchan, outchan]||
  port node-verifyMethod : BiDiPort[inchan, outchan]||
  behaviour: nodeBeh: NodeBeh[node-parent, node-child,
            node-specMethod, node-verifyMethod]
};

define behaviour component type NodeBeh[
  node-parent:  BiDiPort(inchan:[Specification],
       outchan:[Specification]],
  node-child:  BiDiPort(inchan:[Specification],
       outchan:[Specification]],
  node-specMethod:  BiDiPort(inchan:[Specification],
       outchan:[Specification]],
  node-verifyMethod:  BiDiPort(inchan:[Specification],outchan:[Result]]
]
{
  spec: Specification,
  product: Product,
  childList: ChildList,
  thisNodeDef:NodeDef,
  thisNodeID: NodeID,
  NodeBeh[node-parent, node-child, node-specMethod, node-verifyMethod] =
  (
   (node-specMethod@outchan<spec> +
     node-specMethod@inchan(spec) +
     decompose[entry:In[thisNodeDef,childList, spec],
          exit:Out[childList,  spec] +
      removeChild[entry:In[childList, nodeIDs], exit:Out[childList] +
      terminate[entry:In[thisNodeID, childList], exit:Out[null]] ).
      (NodeBeh[node-parent, node-child, node-specMethod,
           node-verifyMethod] + $)
  )
};
```

B.3 Specify Method

This method allows the changes to the Specification component with the Hdev Node.

```
! SPEC METHOD COMPONENT
define component type SpecMethod[
 specMethod-Node:  BiDiPort(inchan:[Specification],
      outchan:[Specification]]]
{
  port specMethod-node : BiDiPort[inchan, outchan]||
  behaviour: specMethodBeh: SpecMethodBeh[node-specMethod]
};

define behaviour component type SpecMethodBeh[
  specMethod-node:  BiDiPort(inchan:[Specification],
       outchan:[Specification]]]
{
  spec: Specification,
  product: Product,
  handleSpec: HandleSpec[entry:In [Specification], exit:Out[Specification],
  SpecMethodBeh[node-specMethod] =
    (node-specMethod@inchan(spec)).
      handleSpec[spec].
    node-specMethod@outchan<spec>) .
      (SpecMethodBeh[node-specMethod] + $)
};
```

B.4 Verify Method

```
! VERIFY METHOD COMPONENT

define component type VerifyMethod[
 verMethod-Node:  BiDiPort(inchan:[Specification],
      outchan:[Specification]]]
{
  port verifyMethod-node : BiDiPort[inchan, outchan]||
  behaviour: verifyMethodBeh: VerifyMethodBeh[node-verifyMethod]
};
```

```
define behaviour component type VerifyMethodBeh[
  verifyMethod-node:  BiDiPort(inchan:[Specification],
       outchan:[Specification]]]
{
  spec: Specification,
  result: Result,
  childNodes: ChildNodes,
  handleVerify: HandleVerify[entry:In [Specification, ChildNodes],
            exit:Out[Result],
  VerifyMethodBeh[node-specMethod] =
    (node-verifyMethod@inchan(spec)) .
    handleVerify[spec, childNodes, result].
    node-verifyMethod@outchan<spec>) .
    (VerifyMethodBeh[node-verifyMethod] + $)
};
```

B.5 Node

The Node is created by composing the HDev component with the SpecifyMethod and VerifyMethod components.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! COMPOSE

compose NodeArch{

hdevNode: HdevNode ||
specMethod: SpecMethod ||
verifyMethod: VerifyMethod||
linkNM : BuffBiDi||
linkNV : BuffBiDi

where

attach hdevNode@node-specMethod@outchan to linkNM@putport@inchan
attach hdevNode@node-specMethod@inchan to linkNM@putport@outchan
attach specMethod@method-node@outchan to linkNM@getport@inchan
```

```
attach specMethod@method-node@inchan to linkNM@getport@outchan

attach hdevNode@node-verifyMethod@outchan to linkNV@putport@inchan
attach hdevNode@node-verifyMethod@inchan to linkNV@putport@outchan
attach verifyMethod@method-node@outchan to linkNV@getport@inchan
attach verifyMethod@method-node@inchan to linkNV@getport@outchan
};
```