Concretizing software architectures using a formal language: towards domain-specific languages for formal development based on π-ADL

Zawar Qayyum

THESE / Université de Bretagne-Sud
sous le sceau de L'Université Européenne de Bretagne

pour obtenir le titre de :
DOCTEUR DE L'UNIVERSITE EUROPEENNE DE BRETAGNE
Mention : Sciences et Technologies de l'Information et de la Communication
École Doctorale Informatique et Applications

présentée par
Zawar QAYYUM

Préparée au laboratoire VALORIA
de l'Université de Bretagne-Sud

Concrétisation des architectures logicielles à l'aide d'un langage formel : vers les langages dédiés au développement formel fondés sur π-ADL

Thèse soutenue le 11 février 2009
devant le jury composé de :

Dr Khalil DRIRA
Chargé de Recherche CNRS, HDR, au LAAS, Université de Toulouse */ rapporteur*

Pr Nicole LEVY
Professeur à l'Université de Versailles Saint-Quentin-en-Yvelines, PRISM, Versailles / *rapporteur*

Pr Yamine AIT-AMEUR
Professeur à l'Université de Poitiers, LISI, Poitiers / *examinateur*

Pr Mourad OUSSALAH
Professeur à l'Université de Nantes, LINA, Nantes / *examinateur*

Pr Pierre-François MARTEAU
Professeur à l'Université de Bretagne-Sud, VALORIA, Vannes / *président du jury*

Pr Flavio OQUENDO
Professeur à l'Université de Bretagne-Sud, VALORIA, Vannes / *directeur de thèse*

Dedicated to my loving mother and the memory of my father…

Acknowledgements

I would like to thank my Supervisor Prof. Flavio Oquendo for introducing me to the field of Architecture Description Languages; for guiding me through the relevant research that enabled me to develop a correct approach to this work; for teaching me how to effectively describe my research efforts in writing; for helping me develop an academic sense and context for my work, expressing it in terms of valid research questions; and his tremendous help at every stage of my PhD. I consider my doctorate as his unconditional gift to me.

My gratitude and thanks to Prof. Khalil Drira, Prof. Nicole Lévy, Prof. Mourad Oussalah, Prof. Yamine Aït-Ameur, and Prof. Pierre-François Marteau for their detailed review, comments and interest in my thesis, and lending the credence of their high academic standings in their evaluation of my work.

This PhD was fully sponsored by the Higher Education Commission of Pakistan, under a visionary policy of enabling the next generation of the country with the skills and knowledge needed for the sustained development of the nation. This program was made possible with the commendable collaboration of the Science and Education department of the French Embassy in Pakistan, under the auspices of the French foreign ministry. These institutions made possible my social insertion into the French society with administrative support and the sponsorship of French language coursework.

Special thanks go to the Archware team that has, over the years, painstakingly developed the complete specification of the π-ADL. This was of great value in my compiler design work.

I also wish to thank Dr. Antoine Beugnard and Dr. Fabien Dagnat of ENST, Université de Bretagne-Occidental, who taught coursework in formal and informal software modelling, allowing me to understand the antics of type systems and formal verification.

My thanks to the members of the Undernet C++ community of highly competent if eccentric IT professionals, particularly Peter Huene who, thanks to his former work with Microsoft, provided me with invaluable help on the Microsoft Intermediate Language; and Adam Wright who helped me in gaining the habitude of stack based programming, and reviewed early designs of the π-ADL.NET compiler.

I would also like to thank Marcos Lopez for his invaluable help in testing the π-ADL.NET compiler and providing the opportunity for the first major application based on this tool, and his patience when the compile time for his code exceeded 15 minutes (we later optimized it to less than 10 seconds). He collaborated with me in the work reported in Chapter 6, and the problem domain of Service Oriented Architectures belongs to his area of interest.

Finally I acknowledge with gratitude the support of my wife during my research work, taking care of all the other things while I focused on this work. And thanks to my little daughter Durriya who has been a source of joy and relaxation that greatly compensated for the stresses of graduate work. And my immense gratitude to my mother who fully supported my decision to pursue higher studies abroad, and is far away in Pakistan while I commit myself to my research in the Occident.

Résumé

L'architecture logicielle est devenue un thème scientifique majeur de l'informatique. En effet, l'architecture logicielle fournit l'abstraction qui permet de développer rigoureusement et de faire évoluer des systèmes logiciels complexes au vu des besoins tant fonctionnels que non fonctionnels. Afin de modéliser les architectures logicielles, un nouveau type de langage est apparu : les langages de description d'architectures (ADL, Architecture Description Language).
Divers ADL ont été proposés dans la littérature, mais ces ADL sont restreints à la modélisation d'architectures abstraites, indépendantes des plateformes d'implémentation. Lors de l'implémentation, l'architecture n'est plus représentée.
Cette thèse s'inscrit dans le domaine des ADL et porte sur la définition et la mise en œuvre d'un langage pour la concrétisation, c'est-à-dire l'implémentation explicite, d'architectures logicielles.
Elle adresse le problème de la construction d'un tel langage et son système d'exécution. Pour cela elle aborde le problème d'un point de vue nouveau : la construction d'un langage centré sur l'architecture logicielle.
Assis sur des bases formelles, notamment sur le π-calcul et π-ADL, ces travaux ont donné lieu à un langage formel pour décrire et garantir l'intégrité architecturale d'un système au niveau de sa spécification, de son implémentation et de ses évolutions ultérieures. La machine virtuelle et le compilateur associé sont enfouis dans la plateforme .NET.

Abstract

Under its various manifestations, the software development life cycle is characterized by the stages of analysis, design, development, testing and maintenance. Various techniques and notations have been proposed for tackling the first three stages, in the interest of establishing an engineering approach to software development. Architecture Description Languages or ADLs are languages designed to model software architectures. The architectural approach suggests the definition of a high-level model of the software system, which can go through multiple stages of refinement to a point where the model is capable of generating implementation code.

Most existing ADLs are primarily focused on describing software architectures, and therefore are not well suited for use as implementation platforms. Particularly, the code generated is incomplete and addresses certain but not all aspects of the target system. To address the problem of decoupling between model and implementation, in this dissertation I present research work that poses π-ADL as a programming language while predominantly retaining its identity as an architecture description language.

To accomplish the integration of π-ADL into the implementation side of the software development life-cycle, a compilation and analysis environment named π-ADL.NET is provided. The design problem of representing a formally-founded, process-oriented ADL in terms of the object oriented .NET platform is explored in detail in the context of our implemented solution. This development effort and subsequent testing is documented in the form of a reusable artefact for developing other language processing tools for π-ADL or other π-calculus based languages.

The π-ADL.NET development effort opened the possibility of π-ADL benefiting from the vast libraries of software components available for the .NET platform. Consequently this research also proposes platform specific extensions to π-ADL for the .NET platform, formally expressed in terms of existing π-ADL syntax. Finally, this dissertation presents a comprehensive case study in modelling service oriented architectures using π-ADL, and discusses a few language adaptations focused on the modelling needs of this domain.

Table of Contents

Chapter 1: Introduction 1

1.1 Software Architectures *1*

1.2 ADLs *2*

1.3 Research Question *3*

1.4 Approach *3*

1.5 Contributions *3*

1.6 Dissertation Structure *4*

Chapter 2: State of the Art 5

2.1 ADLs *5*

2.1.1 MetaH 5
2.1.2 Rapide 7
2.1.3 Weaves 7
2.1.4 C2 8
2.1.5 Wright 9
2.1.6 ArchJava 10
2.1.7 xADL 2.0 11
2.1.8 π-ADL 12
2.1.9 Comparison of ADLs 15

2.2 Model Driven Architecture *15*

2.3 The B Formal Method *16*

2.4 Conclusion *17*

Chapter 3: Background 18

3.1 The π-Calculus *18*

3.1.1 Structural Congruence 18
3.1.2 Reduction Semantics 19
3.1.3 Example 19

3.2 The .NET Framework *19*

3.2.1 The .NET Architecture 20
3.2.2 Support for new .NET Languages 21

Chapter 4: The π-ADL Compiler for .NET 22

4.1 The π-ADL.NET Compiler Architecture *22*

4.2 The Scanner *23*

4.3 The Parser *25*

4.3.1 Compiling π-ADL Parallel Processing Constructs 26
4.3.1.1 Behaviour implementation 26
4.3.1.2 Implementing compose 26
4.3.1.3 Implementing choose 26
4.3.1.4 Implementing replicate 27
4.3.2 Compiling π-ADL Connections 28

4.4 Constructed Data Types *30*

4.4.1 The PiConstructedType class 31
4.4.2 The View class 31
4.4.3 The Polymorph class 32
4.4.4 The Union class 33
4.4.5 The Sequence class 33
4.4.6 Analysis 34

4.5 Syntax Deviations 34

4.6 Chapter Conclusion 35
4.6.1 Related work 35

Chapter 5: Testing Results for π-ADL.NET 37

5.1 Parallel Processing Constructs 37

5.2 Connections 37

5.3 Constructed Data Types 39
5.3.1 Tuple 40
5.3.2 View 40
5.3.3 Union 41
5.3.4 Any 42
5.3.5 Sequence 42

5.4 Conclusion 43

Chapter 6: .NET Extensions to π-ADL 44

6.1 The Extensions 44
6.1.1 Declaring and Instantiating Classes 44
6.1.2 Namespaces 45
6.1.3 Constructors 45
6.1.4 Methods, Fields and Properties 45
6.1.5 Events 47
6.1.6 Enumerations 47
6.1.7 Exceptions 47
6.1.8 Null Values 48
6.1.9 Delegates 48
6.1.10 Casting to and from System::Object 48
6.1.11 Upcasts and Downcasts 49
6.1.12 Generics 49
6.1.13 Inner classes 50

6.2 Case Study 51
6.2.1 Declare and initialize UI components 51
6.2.2 Associate event handlers 53
6.2.3 Handling events in business logic 53
6.2.4 Channeling data into DAO 53
6.2.5 Perform Database Insert 54
6.2.6 Escalating messages to the user 55
6.2.7 Analysis 55

6.3 Related Work 55

6.4 Conforming to Architectural Properties 56

6.5 Conclusion 56

Chapter 7: A π-ADL.NET Case Study in Service Oriented Architectures 57

7.1 Background Concepts 57
7.1.1 MIDAS 57
7.1.2 Building Service-Based Architectural Models 58
7.1.2.1 Service Aggregators 58
7.1.2.2 Service Description and Properties 59
7.1.2.3 Service Relationships 59
7.1.2.4 Taxonomy of Services 59

7.2 The Case Study 60

7.3 The π-ADL Implementation of the SMPP Gateway 61
7.3.1 The Reception Subsystem 61

7.3.2 The SMS Processing Subsystem 66

7.4 Conclusion *67*

Chapter 8: Conclusion 69

8.1 Evaluation *69*
8.1.1 The π-ADL.NET Compiler 69
8.1.2 Testing π-ADL.NET 69
8.1.3 .NET Extensions to π-ADL 70
8.1.4 The SOA Case Study 70

8.2 Future Work *70*
8.2.1 Short term projects 71
8.2.2 Medium term projects 71
8.2.3 Long term projects 72

8.3 Concluding Notes *73*

Appendix I: π-ADL.NET BNF Specification 75

Appendix II: Code Listings of View, Union and Sequence Data Types Runtime Implementations for π-ADL.NET Executables 77

II.1 PiConstructedType *77*

II.2 View *78*

II.3 Polymorph *83*

II.4 Union *84*

II.5 Sequence *90*

Appendix III: π-ADL Model of an SMPP Gateway 97

References 105

Abbreviations 110

Chapter 1: Introduction

Over the past 65 years, the state of the art in software development has progressed from binary machine language on sequentially processed punch cards to the object and component paradigms of today, web applications, domain specific languages, virtual machines, and just in time compilers. Yet this has mostly been a game of catch up. Initially the functional goals of software were limited, and the programmers were often their only users. Program sizes were relatively small, and user interfaces consisted of text printouts. And software programs were standalone, executing in a single thread of execution.

Today's software applications have a much wider audience than their earliest counterparts. They are often distributed across different computers. User interfaces have evolved considerably, and an application may have to cater to different graphical interfaces as standardized for desktops, web browsers and mobile devices. Many applications are required to interoperate with others, which may be built and executed upon heterogeneous platforms. As the audience of software systems broadens, there are additional issues of multi-cultural support, concurrency, and software security.

Thus associated with the transition in general software requirements is a transition in how the software development problem is approached. In order to execute today's large scale and complex software projects developed by large teams of developers, a rigorous and systematic engineering approach is mandatory. For the typical software development life cycle (SDLC) involving analysis, design, development, testing and maintenance, various approaches are available for transforming a specification into an implementation. Architecture driven software engineering is one such approach.

1.1 Software Architectures

The subject of software architectures concerns itself with the high-level view and analysis of software systems, and facilitates a top-down approach to system design. Being a complex property of a software system, which itself can take infinite forms, there is little consensus over an exact definition of a software architecture. According to the IEEE/ISO 1471 standard [IEE00a], **architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution**. [PW92] defines a model for software architecture as follows:

Software Architecture = {Elements, Form, Rationale}

Accordingly, architectural *elements* taking a particular integral *form* define an architecture. The *rationale* encompasses the various choices made in defining the architecture, and can be seen as a set of constraints applicable to functional and non-functional properties of the system model. A software system is said to retain its architectural integrity if it has the following three characteristics [LV95]:

- **Decomposability**: for each component in the architecture, a corresponding unique component exists in the system
- **Interface Conformance**: each system component must conform to its component interface as represented in the architecture
- **Communication Integrity**: the system components communicate only as specified by the interface interconnection in its architecture

The study of software architecture is applied to the following main software engineering issues [GP95][LKJ07][LL99][LLR04] [PW92][SC06]:

- To define a multi-aspect software model that represents the requirements specification in its entirety for the purpose of an implementation that fully conforms to the requirements.

- To provide formal foundations based on which the architecture description can be proven to fulfil certain properties.
- To develop a catalogue of reusable methods, techniques, patterns and rules that would aid in architecture description.
- To propose domain specific solutions that enable limited generalizations applicable to families of architectures. This results in architectural styles.
- To provide the ability to maintain or evolve an architecture after it has been implemented.
- To customize an existing architecture in order to apply it to a different problem or problem domain.
- To verify, either statically or dynamically, whether the implementation conforms to the architecture.

As a discipline, software architecture has evolved from an empirical categorization of practically validated design techniques into a broader, more organized coverage of tools and concepts, providing concrete and scalable guidance for all aspects of the software design and development process [SC06]. Instrumental to this evolution were architecture description languages or ADLs.

1.2 ADLs

An ADL can be loosely defined as a language that focuses on the high-level structure of the overall application rather than the implementation details of any specific source module [MT00]. They provide a consistent and sometimes formal notation for defining architectures, providing a common representation for different aspects of architecture. ADLs permit analysis of architectures in terms of completeness, consistency, clarity, and performance, and can support automatic generation of implementation code to a certain degree.

Amongst the numerous ADLs proposed, divergent trends are often obvious. Some ADLs focus on ease of understanding and communication for developers, by supplying an intuitive, sometimes visual syntax to aid user comprehension. Other ADLs concern themselves with formally founded syntax so that certain properties of architectures can be mathematically analyzed. Certain ADLs propose a fixed modelling style with visible constraints on the mechanics of component interconnection. Others are more relaxed, and even style agnostic. As each ADL attempts to advance the field by addressing new design concerns or already covered design areas using a more effective technique, it elongates the taxonomic space of ADLs in that direction, making it more difficult to exactly define what an ADL is. Consequently there is little consensus as to the exact definition of an ADL [MT00]. That is why our loose definition above tends to stay away from disputed territory by structuring the description of an ADL around our conception of an architecture.

As a consequence of their primary focus on high-level design, ADLs in general are unsuitable as implementation platforms. For example, in describing the fragmentation of the software architecture community, [MRR02] divides it into the research and the practitioner groups. In a simplified comparison, it postulates that while the research community emphasises formal rigor and depth over breadth, the practitioner community focuses on practicality over formality and breadth over depth. In the former case, the architecture fails to include all aspects of the implementation. In the latter case, the lack of formal rigour entails a difficulty in automatically translating design into implementation, and in verifying conformity between the two. As argued in [Ald03] the problem of correctly converting the architecture into implementation will remain as long as the ADL is different from the implementation language. This issue is vital to formulating the research question of this dissertation (next section).

The ADL that forms the focus for this work is the π-ADL reported in [Oqu04]. π-ADL is a second generation Architecture Description Language which distinguishes itself from first generation ADLs on the following grounds: it is formally founded, in this case on the π-calculus, and it provides syntax for customizing architectural styles. Based on this extra rigor and versatility, π-ADL can be seen as an ADL that better meets the challenges of architecture driven software engineering than first generation ADLs. The language syntax is designed incrementally with higher level functionality built upon basic syntactic layers. The advantage to this approach is that it opens the possibility of extensions to the language, allowing for domain specific syntactic enhancements. The approach also allows greater leeway for syntactic experimentation and evolving the language specification.

1.3 Research Question

The fundamental question of the research presented in this dissertation can be defined as follows:

Is it feasible to develop an ADL that supports the phases from architecture description to implementation, in order to:

- *Preserve the architectural integrity of the system at the implementation level;*
- *Support analysis of the concrete architecture;*
- *Support evolution of the implementation while enforcing its architecture integrity;*
- *And directly use the implementation mechanisms of the hosting platform?*

1.4 Approach

The work presented in this dissertation provides an answer to this research question by integrating an ADL into the implementation side of the SDLC, so that effectively the architecture described using the ADL becomes the implementation. This is accomplished by transforming an ADL into a programming language by:

a) providing an implementation level compiler and analysis tool for the ADL,
b) incorporating some modifications in the ADL syntax in order for it to fulfil the role of a deterministic implementation language while retaining its formal ADL comportment, and
c) proposing formally founded platform specific extensions to the ADL in order to enable it to access reusable software components available on the implementation platform. The resulting compiler environment is called π-ADL.NET.

The ability to validate and compile an ADL on a mainstream platform has certain additional advantages. Firstly, it opens the possibility of integrating the ADL code with the code written in a detail oriented language, such has C# or Visual Basic.NET, since they are both compiling to the same target platform. That way, the software architect's investment in the design effort is employed directly in the resultant software solution. Secondly, the compiled ADL code can access and utilize the large number of reusable software libraries already developed for the platform. Third, it is an interesting approach to heterogeneous software development, whereby different portions of a software system are programmed in languages better suited to their development. For example in implementing a large software project, an ADL can be used for the high-level architectural specification, and a 3G language be used for the detail oriented leg work.

1.5 Contributions

In this research work I have made the following contributions:

π-ADL.NET compiler

- A functionally verified mapping of π-ADL to CIL. This mapping required the use of certain .NET components along with the CIL syntax, for the threading and concurrency features of π-ADL.
- A Scanner, Parser and Code Generator suite that constitutes the .NET compiler for π-ADL. This software is comprised of over 10000 lines of C# code.
- A compilation and execution GUI that allows the programmer to access the .NET compiler for π-ADL.

π-ADL extensions for .NET

- A set of π-ADL extensions that permit complete access to existing .NET software components, written in any other .NET language. Syntax for the ability to generate such components using π-ADL is not proposed.
- A formal foundation for these extensions, expressed mostly in terms of core π-ADL syntax, or the underlying π-calculus where necessary.

Summary of test cases for π-ADL.NET

- A summary of development, testing and usage experience organized as test cases that can be generalized for other π-ADL language processing tools, or for processing other π-calculus based languages.

Case Study of the π-ADL.NET compiler

- A case study in π-ADL to implement a service oriented architecture for an SMS distribution system, with a notable feature of dynamic service discovery.

1.6 Dissertation Structure

The rest of this dissertation is structured as follows:

State of the Art: This chapter presents the current trends in the research and practice of architecture-based software development, and relates how my work covers some open areas in the field.

Background: In this chapter, the background research that led to this work is presented. It presents π-calculus in sufficient detail in order to describe the formal basis of π-ADL. An overview of the .NET technology platform is given along with its useful features as foundation for my research work.

The π-ADL Compiler: This chapter provides implementation details of the π-ADL.NET compiler developed as part of the research work. The compiler architecture is presented along with the functional mapping of π-ADL upon CIL. This discussion covers the rationale behind the design choices and attempts to identify a general direction for the approach taken, where applicable.

Testing π-ADL.NET: This chapter presents a set of test cases used to test π-ADL.NET. It sheds light on the scope of the project and on how the language structure influenced π-ADL.NET architecture.

π-ADL Extensions for .NET: In this chapter I present .NET platform specific extensions to π-ADL in order to allow it to use existing .NET software components compiled using any other .NET language. These extensions are tested using a case study in the standard three-tier desktop software architecture.

Case Study in Service Oriented Architectures: This chapter presents a case study in the description of service oriented architecture using the π-ADL.NET compiler. The functional capability of the π-ADL.NET compiler is demonstrated in the context of modelling a SMS distribution system with a provision for dynamic service discovery.

Conclusion: In this chapter, the research work is reviewed and its accomplishments summarized vis-à-vis the stated goals. There is a discussion of the possibilities of future research work that can utilize what has already been completed.

Chapter 2: State of the Art

This chapter serves to place π-ADL.NET in the proper research context in order to highlight the specific contributions and achievements of this project, and differentiate it from previous work. Here we attempt to see how π-ADL.NET fares vis-à-vis other solutions available in research and practice on this subject.

2.1 ADLs

In order to effectively model software architectures at a generic or domain-specific level, various architecture description languages or ADLs have been proposed. Some of the ADLs reported in literature are π-ADL [Oqu04], ACME [GMW97], ADML [Spe00], Aesop [GAO94], ArchJava [Ald03], Armani [BJC05], C2 [MOR96], CommUnity [LWF03], Darwin [MDE95], MetaH [LCV00], Pilar [CFB02], Plastik [JBC05], PRISMA [APR05], Rapide [LKA95], SADL [RMQ95], UniCon [SDK95], Weaves [GQ94], Wright [AG94], and xADL 1.1 [KGO01] and 2.0 [DVT01]. Due to the domain specific nature of certain ADLs, their approach depends on how the language designers perceive the semantic needs of the family of software architectures they propose to model. From the point of view of impact on the software development life cycle, different ADLs provide different levels of support for specification, refinement and implementation of software systems. Here I perform a general survey of a representative set of ADLs, and then focus on the level of implementation support each one of them provides. For each ADL thus treated, the following information is covered:

- The class of systems it proposes to model
- The way a software architecture is conceived by its designers
- Formal foundations, if any
- Support for specification, refinement and implementation
- Analysis of its strengths and weaknesses

2.1.1 MetaH

Domain: MetaH [LCV00] is an ADL applied predominantly to model embedded real-time systems proposes a highly execution centric approach that spans the domains of both software and hardware. The language was designed at Honeywell for the specification of real-time, fault-tolerant, securely partitioned, dynamically reconfigurable multi-processor system architectures. It has been used to describe system architectures for mission critical applications such as weapon systems and aircraft.

Conception: MetaH proposes an integrated system architecture encompassing both software and hardware elements, and is structured in terms of components and connectors that associate using various interface types. Components can be primitive or non-primitive. Primitive components, which are classified as event, port and type, are used to describe the properties of non-primitive components, which represent hardware or software elements in the implementation. Some examples of non-primitive components are listed in table 2.1:

Component	Type	Description
Subprogram	Software	Function or procedure
Process	Software	A schedulable process with its own memory
Package	Software	A collection of subprograms and persistent data objects
Monitor	Software	A package that additionally enforces synchronous access to its data
Processor	Hardware	Entity that executes software processes
Device	Hardware	A non-processing entity that can communicate via ports and

		provide processors access to its memory
Memory	Hardware	Block of memory that can be shared by processors
System	Hardware	Collection of processors
Application	Hardware & Software	Represents the highest-level component that encompasses all other elements in the architecture

Table 2.1. Non-primitive components in MetaH.

Formal Foundations: MetaH is not founded upon any formal system of reasoning, and is based upon an intuitive component-connector paradigm.

Tool Support: MetaH is supported by a wide range of graphical and textual tools. There are modellers and analyzers for schedulability, reliability and security, a binder for hardware and software, an executive generator, an application builder and a workspace that integrates these tools. These tools are available for generating partial or domain-specific implementation code based on the architecture description, and to support change propagation. We consider these tools below [KVL98][Ves98], grouped respectively for their modelling, refinement and implementation roles:

- *Modelling and Analysis*: The Schedulability Modeller and Analyzer is concerned with the real-time performance characteristics of the processes in an application. The central property being modelled or analyzed is the schedulability of the application i.e. whether each process in the application will complete its execution within the specified time constraints for each scenario. The analysis reports the range of flexibility in component execution times while retaining the application as schedulable, considers worst-case scenarios, and provides a framework for improving the architect's estimates of process deadlines.
  The Reliability Modeller and Analyzer allows the analysis of the application fault tolerance by modelling randomly arriving fault events, component states and the set of fault responses. Component attributes relevant to the fault model include fault event and propagation rates, error-paths and error-management protocols. Based on the reliability specification, two types of statistical models can be generated: a Markov-chain based model, and a finite state machine model for concurrent processes. The latter can be analyzed with in-built tool support, while the former can be analyzed using third-party tools.
  The Security/Safety Modeller and Analyzer allows the definition of safety levels for components. The analysis permits a view of security integrity in a component in the event of failure in other interacting components.
- *Refinement*: MetaH has strong support for refinement provided by the Software/Hardware Binder, which allows the binding of a software component with a hardware component. Processes are bound to individual processors, data transfer connections are established over communication channels, and reusable code and data components are bound to memory components. For unbound or partially bound components, the tool automatically creates a binding with one of the available resources.
- *Implementation*: Based on the architectural model, a skeleton implementation can be generated by the Executive Generator and Application Binder. This tool generates component definitions, their interconnection code, and a system executive. Furthermore, this architecture based implementation code is customized for the target hardware environment. The implementation support includes the generation of target implementation language code and compiler directives. A Source Code Repository exists, from which component code can be linked into the generated implementation.

Analysis: MetaH can be categorized as a domain-specific ADL cast in a particular architectural style, with extensive support for modelling and refinement, and partial support for implementation. Since the application domain for the language is well defined and grounded in the interaction of software and hardware components, the application of the language to a wider range of software systems cannot be envisioned without fundamentally modifying it. While the tool support for modelling, analysis, and refinement is remarkable, the MetaH language and consequently the development environment lack the foundations for generating detailed implementation code.

2.1.2 Rapide

Domain: Rapide [LV95][LKA95] is an ADL conceived for the purposes of modelling distributed systems and takes a comprehensive, all-inclusive approach to architecture specification in order to cover the many different facets of such a generalised class of systems.

Conception: Rapide is presented as a family of five different domain specific languages for describing components, their interfaces, event flow and patterns, executable behaviours and constraints. A Rapide architecture is an abstract model of system execution, referred to as an interface connection architecture [LVM95]. It consists of *interfaces*, *connections* and *constraints*. Interfaces specify the component behaviour in a system, connections interconnect components, and constraints are applicable to both components and connections. The language design is well conceived with attention to fundamental architectural concepts of component and communication abstraction and their causal interrelationship, communication integrity, dynamicity, hierarchical refinement and inter-transformability between multiple levels of detail.

Formal Foundations: The central semantic approach taken by Rapide is to present a software architecture in terms of a partially order sets (posets) of events and multiple processes that observe it. The poset foundation allows formal reasoning on the sequential interrelationships of events. The ensemble of events represents a set of execution traces that work as a global constraint system for processes.

Tool Support: The tool support for Rapide primarily consists of a compiler with a library management system (rpdc), and a partial order viewer (POV). The compiler translates a Rapide specification into an executable module or a usable library. The integrated library management system allows storage, management and retrieval of compiled libraries, and facilitates reuse of such libraries when generating other architecture modules. The executable generated can be thought of as an executable model of the architecture, and as such does not encompass implementation level details.
The POV tool operates on log files generated by the execution of a Rapide program, and uses the execution trace to generate a poset that can be graphically browsed. It also facilitates filtering by event types in the poset to narrow down on the events of interest.
Although the proposed Rapide modelling process [Luc96] suggests the integration of a constraint checker and animation tool into the development process, as of this writing these tool have not been provided. Consequently, Rapide can be seen as predominantly a modelling language with no considerable implementation support.

Analysis: Despite its formal foundations, broad application domain and comprehensive modelling syntax, Rapide has little tool support [Luc96], which is limited to generating executable simulations of Rapide based architectures and analysing them.

2.1.3 Weaves

Domain: Weaves [GQ94] refer to a class of system models made up of a large number of software components, referred to as *tool fragments* [GR91], which exchange arbitrary data objects, the information content of which can be structured in a complex schema. Weaves are primarily design to foster the smooth, incremental construction of *observable* systems [GR91]. An observable system in the Weaves sense is one that gives adequate exposure to state information of its components and data. Although the creators of weaves present it as an architectural style and not as an ADL, the underlying modelling language is generally also referred to as weaves in literature [MT00], and we use the same name for both.

Conception: A tool fragment is a small software component that performs a single, well-defined function. It can be thought of as a method in Java, except that it is stateful. Tool fragments are consumers and producers of objects. Each tool fragment executes as an independent thread. Objects are transmitted from one tool fragment to another via ports attached to queues. The function of queues is to buffer and synchronize communication among tool fragments. The objects traversing a weave, and the ports and queues through which they travel, are passive. Only the tool fragments are active, accepting objects from ports, invoking the object methods, performing relevant computations, and passing objects downstream.

Formal Foundations: Formally, the weaves interconnection model represents a bipartite, directed multigraph.

Tool Support: The construction of weaves is accomplished using the Weaves Visual editor Jacquard. The goal of the Jacquard visual editor is to provide a visual engineering environment that allows the complete description of a weave, with special emphasis on facilitating the maintenance of large and complex models. In order to do this, it incorporates concepts such as zooming, parallel contextual information access, and browsing capability. We briefly describe each of these below:

- *Zooming*: Zooming is defined as a single visual mechanism that supports all forms of visual browsing and access, and is referred to as drill-down and drill-up functionality in later systems [Jer97]. Its implementation in Jacquard is combined with planar navigation to give the user an appropriate interface for examining finer details of a system component through incremental browsing, without losing perspective of the overall system.
- *Contextual Information Access*: Jacquard attempts to integrate the large amount of component and system documentation into the system model for large scale and complex systems, and supports contextual information access in two ways. Firstly it integrates a component browser *Carlyle* into Jacquard to permit users direct access to online component documentation. Secondly it supports a large variety of annotations that can be embedded in a weave. With these two support forms, the architect can incorporate text, documents, diagrams, pictures, videos and sounds into the architecture description. Furthermore, the information organization enabled with Carlyle is well defined and permits standard component level documentation including functional descriptions, interface definitions, examples etc.
- *Browsing*: Jacquard supports browsing of component libraries and system structures. It provides customizable component trays, which allow the user to organize components in collections specific to a particular domain. The browsing of system structure is accomplished with the help of zooming capability. The details of a tool fragment are open to analysis and modelling, and can be considered as a sub-weave. This allows the definition of hierarchical systems.

Analysis: The Weaves environment introduced some innovative visualization concepts which can be seen as precursors to modern day online analytical processing (OLAP) techniques. It is especially suited for heavily annotated architectures. The engineering environment Jacquard is a comprehensive and well integrated solution for weaves, however it is restricted to modelling and refinement. There is no support for implementation, nor is there a language component for modelling executable behaviour.

2.1.4 C2

Domain: C2 [MOR96] [MR99] refers to the C2 architectural style and its accompanying ADL C2 SADL. It has been designed for the modelling of user interface intensive and highly distributed systems, assuming the following:

- Components written in different programming languages
- Distributed and heterogeneous component environment without a shared address space
- Dynamic architectures
- Multi-user access
- Multimedia interfaces

Conception: The C2 style can be described as a network of concurrent components linked through message routing devices called connectors. The communication details are handled by connectors, while the computations and state information is handled by components. The style is distinct in that it enforces a strict interface paradigm at the component level. Each component has only two interfaces

Up and *Down*, with which it can communicate with other components via connectors. Connector interconnection is also possible, with no limit on the number of interfaces [MRR02][MR99]. This can be viewed as a deliberate design effort to separate communication details from the component definition as much as possible. This implies however that C2 supports a very specific architectural style grounded on this approach.
In order to enforce the C2 style on architecture descriptions, an internal component architecture has also been specified [MT96]. Based on this architecture, a dialog sub system within the component maps external requests and notifications into internal component operations.

Formal Foundations: C2 does not have any formal foundations and is described informally as a network of concurrent components attached together by connectors.

Tool Support: C2 enjoys very little tool support. The ArchShell tool exists for the interactive construction, execution and runtime modification of C2-style architectures. It is a command line application and thus provides a very narrow interface for specifying architectures, with not visualization support. A set of commands allow the creation of an architecture incrementally. The interface allows communication with different components at runtime, and provides dynamic support by loading and linking new architectural elements at runtime.
C2 also has implementation support based on Java or C++ class frameworks. These frameworks come in the form of extensible abstract class libraries defining central C2 concepts such as components, connectors and messages. This allows developers to implement components that conform to the C2 style, and consequently employ them in C2 architectures.
The tool support for the C2 style provides architecture modelling and component integration facilities, with the latter being enabled only in a heterogeneous language context.

Analysis: By restricting architectural style, C2 attempts to provide a reliable, baseline framework for describing highly distributed systems with rich and diverse interfaces. The lack of tool support means that it is difficult to validate this claim practically. Although there is no implementation support, C2 prescribes implementation related restrictions on internal component design.

2.1.5 Wright

Domain: The primary focus of Wright [AG94][All97] is to provide a formal system for specifying architectural connector types. The description of these connector types is based on the idea of adapting communication protocols to the description of component interactions in a software architecture. In fact these protocols are modelled after network transfer protocols. The modelling approach leads to a deductive system in which architectural compatibility can be checked in a way analogous to type checking in programming languages, thus endowing the ADL with a good degree of rigor in type verification [AG94].

Conception: An architecture in Wright can be described in three parts:
- Component and connector types;
- Component and connector instances; and
- Configuration of component and connector instances.

Unlike C2, Wright does not impose any architectural style. However there exist certain constraints, such as inability to connect connectors to each other directly. Similarly components cannot connect to each other. This distinction is encoded syntactically by calling component interfaces *ports*, and connector interfaces as *roles*. Connectors have associated *glue* which specifies the interaction of its roles with connected components.

Formal Foundations: Wright uses a subset of the Communicating Sequential Processes formal language (CSP) [Hoa85] to specify architectural behaviour. The subset is capable of defining finite state processes [MRR02]. In CSP communicating entities interact based on communication events.

The events are capable of input and output of data, and thus take on the specification role defined for functions in functional programming.

Tool Support: Wright is predominantly a design language [Ald03], with no syntactic or tool support for implementation. A parsing and translation toolkit exists for other high level languages [STA08]. This includes support for translating Wright specifications into CSP and ACME. A Wright-annotated specification in ACME can also be translated into Wright. We observe that the need to annotate an ACME-based specification in order to completely represent Wright-based architectures means that tools for pure ACME are not suitable for analyzing all aspects of a Wright translation to ACME.

Analysis: Wright is primarily concerned with formal analysis of certain architectural properties. Due to this overwhelming focus, and lack of tool support for detailed modeling and implementation, the language has limited scope beyond its radius of formal reasoning. However its formal foundations are well developed and exposed, and consequently it has an important role in benchmarking subsequent ADL work concerned with architectural formalization.

2.1.6 ArchJava

Domain: ArchJava [Ald03] is an ADL that extends the java programming language with general-purpose architecture description constructs of components, connections and ports, and provides a complementary type system. The primary semantic contribution of ArchJava is to extend java to enable communication integrity within the language. The approach is unique in the sense that here a programming language is extended with syntax to allow architecture descriptions.

Conception: In order to represent the architecture description constructs, ArchJava uses the following approaches:

- A component is represented by a class qualified with the **component** keyword.
- A port is defined as a curly-bracket enclosed body of code that defines method prototypes. These prototypes are declared using one of the **requires**, **provides** and **broadcast** keywords. A required or a broadcast method must be implemented by a component that connects with the encapsulating component of the port. The difference between these two types is that a component can connect to multiple implementations of a broadcast method, and broadcast methods always return **void**. A provided method is defined by the parent component of the port.
- Ports are connected using the **connect** keyword, followed by the fully qualified port names to be connected over instances of components.

ArchJava supports hierarchical architectures by composing components within a parent component. Component instances are declared inside a component class, and their ports are then connected using the **connect** keyword. It achieves communication integrity by ensuring that components can call each others' methods only through declared connections between ports. Although dynamic component creation is supported, communication integrity in this case is still preserved by disallowing the passage of the component as is outside its component. Note that it can still be passed to other components while cast as an Object type in Java.
A particularly relevant issue for communication integrity in ArchJava is shared data. An object passed from one component to another can contain methods that invoke functionality in the sending or other components that is normally restricted for the acquiring component. ArchJava prevents such surreptitious means of violating this architectural property by disallowing objects from storing component references.

Formal Foundations: The formal type system of ArchJava extends only to the architectural syntax contribution of the language to its Java foundation. Thus ArchJava can be defined as a semi-formal language with limited exposure to formal analysis techniques.

Tool Support: ArchJava has both modelling and implementation support. A compiler for ArchJava has been developed, based on the standard Java compiler. A Graphviz [Gra08] based extension is also provided for viewing graphical representations of ArchJava architectures. ArchJava has IDE support through a plugin for AcmeStudio [Acm08], which is an eclipse based IDE. The plugin allows the design of an ArchJava architecture through the use of a diagram style for the ADL. ArchJava representations can benefit from AcmeStudio features such as constraint and ADL design rule checking. With this modelling and compilation support, ArchJava has the design and implementation cycle covered for any architectures that are defined using it.

Analysis: Being a hybrid, ArchJava is unique in that it acts both as an architecture description and as an implementation language. As an architectural solution, it presents a fresh approach, but at the same time is limited in multiple ways. Architectures represented in ArchJava are more concrete then those expressed in "pure" ADLs, because the language has a strong implementation base. Also, it currently supports just one architectural style of inter-component connections through method calls. Based essentially on an informal Java foundation, an entire ArchJava architecture cannot be subjected to a formal reasoning using a single system, despite the formally well-founded type system of the ArchJava component extensions.

2.1.7 xADL 2.0

Domain: xADL 2.0 [DVT01] is an extensible XML-based ADL built with the purpose of allowing architecture researchers to quickly experiment with new modelling constructs and features in architecture description languages. It represents the current state of evolution in ADLs that use XML as their foundation.

Conception: The distinction between xADL 2.0 and previous XML based ADLs such as xADL 1.1 [KGO01] and ADML [Spe00] is grounded in developments in the way extensibility is catered for in XML. In the XML 1.0 specification [BPS06], the user is provided with a means for defining a new custom tag library by writing a document type definition (DTD). While DTDs allow the user to define portable and reusable extensions to XML that serve a specific application domain, they do not support the concept of inheritance, and lack a type system for defining simple or compound types. XML Schemas [Fal04] address these issues and allow the user to extend existing schemas to create more specialized syntax.
While the earlier XML-based ADLs xADL 1.1 and ADML are defined in XML DTDs, xADL 2.0 is defined using XML Schemas. This endows xADL 2.0 with the advantages of type definitions based on attributes and elements, and their extensibility i.e. defining architectural styles. xADL 2.0 is defined in multiple XML schemas, that can be organized in three logical groups:

- *Architecture Modelling – Description and Prescription*: This group of schemas is concerned with modelling the static and runtime aspects of the architecture. xADL 2.0 makes a distinction between these two aspects, arguing that different sets of properties are relevant to each of these aspects. These schemas define components, connectors, interfaces, links, and flat and hierarchical grouping constructs. Components, connectors and interfaces also have programming language-style type information associated with them, in order to enable meaningful comparisons between their instances in an architectural model.
- *Instantiable Architectures*: The abstract implementation schema in xADL 2.0 provides types that act as abstract place holders for component and connector implementations. This schema can be further extended to define specific syntactic information as it applies to component and connector implementations in different programming languages. The rationale for this aspect of the architecture model is to enable tool support for runtime analysis and manipulation of implementations at an architectural level.
- *Architecture Configuration Management*: This group of schemas allows the application of configuration management concepts such as versioning, options, and variants to architectures and

architectural elements. While it is questionable whether configuration management should be applied at the architectural level, given the typically fine-grained level at which it is usually handled, these xADL 2.0 schemas purport the added benefit of defining *product family architectures*. A set of product family architectures are similar architectures that differ from each other based upon the specific focus of their target implementation.

Formal Foundations: xADL 2.0 does not have any formal foundations, and design concepts embodied in its schemas are described as "semantically neutral" [DVT01].

Tool support: An advocated advantage of an XML based ADL is the availability of many commercial off-the-shelf tools that can support it. xADL 2.0 particularly benefits from XSV [WWW07] and XML Spy [Spy08]. XSV is a schema validator and allows the developer of a xADL 2.0 schema to validate his work for correctness. XML Spy is a graphical tool for creating, editing and visualizing XML schemas. In addition it provides a graphical annotated tree representation of schema types and elements for facilitating user analysis.
Generic XML tools are less relevant when it comes to actual architectural modelling using xADL 2.0. For modelling xADL 2.0 architectures, java APIs have been developed that allow a programmer to write Java code that will generate xADL 2.0 specific XML resulting in an architectural model. These APIs are based on the Document Object Model, which provides an object-oriented interface to XML. It is notable that for new extensions to xADL 2.0, a new set of APIs needs to be coded. Therefore the modelling support for xADL 2.0 is not comprehensive, which should ideally utilize XML schemas complemented with graphical annotations to dynamically construct a visual modelling environment.

Analysis: xADL 2.0 can be described with good reason as a true second generation XML-based ADL, since it's meta-model is encoded using a paradigm which is significantly more descriptive and extensible when compared to previous XML-based ADLs. The "high extensibility" claim however is put into question, since extending the ADL meta-model does not automatically grant tool support for the extensions. With Microsoft DSL Tools [GS03] for example, we see exactly such an implementation, which allows the development of meta-models which can then be used to automatically create faithful modelling environments.

2.1.8 π-ADL

Domain: π-ADL defines a benchmark for second generation ADLs in the sense that it is formally founded and allows the customization of run-time architectural concepts. The second feature implies that the definition of component and connector semantics can be customized, allowing a large amount of flexibility when defining architectural styles [OWM04]. The language has been applied in terms of style definition and modelling for the following diverse domains: ambient intelligence, sensor-actuator networks, mobile agent systems, human-computer interfaces for monitoring systems, grid computing systems, enterprise application integration systems, as well as software systems targeting J2EE platforms.

Formal Foundations: π-ADL is formally founded on the higher-order typed π-calculus (hence the name), which encompasses a formal transition and type system. It conforms to the language design principles of correspondence, abstraction and data type completeness [OWM04]. Type completeness assures first class citizenship to all data types i.e. they can be declared, assigned, can have equality defined over them, and can be persisted.
The structural operational semantics of π-ADL represents behaviour (and thereby computation) by means of a deductive system, expressed by a formal transition system, in line with a type system. Further, type soundness asserts that well-typed terms do not give rise to runtime errors under the transition system.

Conception: The π-ADL is fundamentally organized around the concept of communicating processes. In a π-ADL program, the top level constructs are behaviours and abstractions. Each behaviour definition results in a separate execution entry point, meaning that the program will have as many top level concurrent threads of execution as the number of behaviours it defines. Abstractions are process definitions that can be "pseudo-applied" or instantiated from behaviours and other abstractions. An abstraction is capable of receiving a single argument when invoked.

In π-ADL, an architecture is described in terms of components, connectors, and their composition. Components are described in terms of external ports and an internal behaviour. Ports are described in terms of connections between a component and its environment. Protocols may be enforced by ports and among ports. Connectors in π-ADL are special-purpose components. Their architectural role is to connect components together.

Components and connectors can be composed to construct composite elements, which may themselves be components or connectors. Composite elements can be decomposed and recomposed in different ways or with different components in order to construct different compositions [Oqu04a].

Architectures are composite elements representing systems. In π-ADL, architectures, components, and connectors are formally specified in terms of typed abstractions over behaviours. The π-ADL.NET implementation currently doesn't support this level of typing, and at the highest level supports behaviours and abstractions.

The body of a behaviour or an abstraction can contain variable and *connection* declarations. Connections provide functionality analogous to channels in π-calculus: code in different parts of behaviours or abstractions can communicate synchronously via connections, and connections can also interconnect behaviours and pseudo-applications of abstractions. Connections are typed, and can send and receive any of the existing variable types, as well as connections themselves. Sending a value via a connection is called an output-prefix, and receiving via a connection is called an input prefix, as both of these operations are comparable to the π-calculus constructs of the same name. Listing 2.1 shows a simple program in which a behaviour pseudo-applies an abstraction, and associates its connection *x* with the connection *y* of the abstraction through the *rename* clause. This enables communication between the behaviour and the abstraction during the course of their respective executions.

Program a: Behaviours, Abstractions and Connections

```
behaviour {
  x : Connection[Integer];
  compose  {
    via myAbs send 42 where {x renames y};
  and
    via x send 101;
  }
}
value myAbs is abstraction (argi : Integer) {
  y : Connection[Integer];
  i : Integer;
  via y receive i;
  argi = i * argi;
}
```

Program b: Compose and choose

```
behaviour {
  x : Connection[Integer];
  y : Connection[Boolean];
  a : Integer;  b : Boolean;
  compose  {
    via y send true;
  and
    choose {
      via x receive a;
    or
      via y receive b;
    } //end choose
  } //end compose
} //end behaviour
```

Listing 2.1. Two π-ADL programs demonstrating various language concepts.

The *compose* keyword seen in Listing 2.1 serves the purpose of creating two or more parallel threads of execution within a program and corresponds to the *concurrency* construct in π-calculus. The generalized syntax for a compose block is:

```
composeBlock := "compose {" block [" and " block]+ "}"
```

where each block inside the compose block results in a separate thread of execution. Note that in both cases, if the two statements inside the compose block were coded to execute in a single thread, a deadlock would have occurred.

Another important π-ADL construct is the *choose* block. It has the following generalized syntax:

```
chooseBlock := "choose {" block [" or " block]+ "}"
```

Only one of the sub-blocks inside a choose block is executed when execution passes into the choose block. For example, in Listing 2.1, only one of the two choose sub-blocks can execute. Since a value is available via y and not via x, the second sub-block will execute and the first sub-block will be terminated. When more than one sub-block are eligible for commencing execution at the same time, the selection criteria for the block to be executed is non-deterministic and not defined in the language.
To provide the equivalent of the π-calculus replicate construct, π-ADL supports the *replicate* keyword, with the following syntax:

```
  replicateBlock := "replicate {" block "}"
```

Semantically, this entails that the contents of the replicate block are infinitely replicated in parallel threads of execution. As we will see in Chapter 4, the implementation of replicate has been modelled with the limits of real world computers kept in mind.
For a high-level architecture description language, π-ADL represents a large and varied set of data types, in addition to the primitive types Integer, String, Boolean and Float. These constructed data types are: Tuple, View, Any, Union, Sequence, Set, Bag, Quote, Variant, and Location.
The current implementation of π-ADL.NET implements only the first five data types in the above list. A short description of each of these data types is as follows:

- **Tuple**: The values of a tuple type `tuple[`T_1`, …, `T_n`]`, for n≥2, are n-tuples tuple(v_1, …, v_n) where each v_i is of type T_i. The individual values within the tuple can be projected into other variables using the `project` syntax e.g. `project t as a, b;`.
- **View**: Views are labeled forms of tuples. The values of a view type `view[label`$_1$` : `T_1`, …, label`$_n$` : `T_n`]`, for n≥2, are views view($label_1$: v_1, …, $label_n$: v_n) where each v_i is of type T_i. In addition to projecting like a tuple, a view also supports short-hand projection, much like accessing a class variable in Java or C# e.g. in order to reference the member `a` of view `v`, we use the syntax `v::a;`.
- **Union**: A union type `union[`T_1`,…,`T_n`]`, for *n*≥2, is the disjoint union of the types T_1, …,T_n, with values union(T_i::v) where v is of one of the types T_i. The current type of a union can be determined by processing it in a select-case block of code, where each case is represented by one of the member data types of the union. There is also a type-equality Boolean operator `#=` that compares the current type values of two unions for equality.
- **Any**: An any type is like a union type with no constraint on the type of value it can hold.
- **Sequence**: A sequence is a collection in which the elements are ordered. An element may be part of a sequence more than once. The values of a sequence type sequence[T] are sequences sequence(v_1, …, v_n) where each v_i is of type T.

The usage of constructed data types is demonstrated in the case studies documented in this dissertation.

Tool Support: Within the context of the Archware project [OWM04], an integrated development environment was developed for π-ADL and related languages. It is comprised of the following functional components:

- The ArchWare Core Environment which provides the Archware ADL Compiler and Virtual Machine that supports the enactment of architecture descriptions.
- The ArchWare Core Meta-Process Models which provide the support for software processes that are used to build and evolve software applications.
- The ArchWare Environment Components which provide the ArchWare tools that support architecture description, analysis, and refinement processes.

Together, these tools allow the compilation of architectural models into their executable representations, manage a store of reusable process abstractions, and provide a mechanism for evolving software architecture components using a graph-based visual representation.
The Archware IDE was developed as a research tool with the goal of evaluating concepts pertinent to

executable architectures and their evolution. The π-ADL.NET project, described throughout this dissertation, targets a different set of research issues, namely the transformation of these executable architectures into implementation equivalents, and physically grounding them into a wide technological base that enables architectural experiments in diverse application domains.

Analysis: π-ADL is a well designed and well rounded language, bringing the advantages of formal foundations, style customization, small syntactic base and ease of use to the study of architecture description. The style definition syntax is particularly useful, but unfortunately does not yet have tool support. The π-ADL.NET project, complemented by visual notations for base and style-specific depiction of architectural concepts, can overcome this limitation, and can help to propel π-ADL to the forefront of architecture modelling and experimentation.

2.1.9 Comparison of ADLs

In terms of original contribution, π-ADL.NET compares well with the other ADLs discussed here. As can be seen in the comparison table 2.2, it is the only ADL that is fully formally founded, is domain independent, and provides comprehensive tool support for modelling, refinement and implementation of an architecture.

ADL	**Formal foundations**	**Domain Independent**	**Tool Support**		
			Modelling	**Refinement**	**Implementation**
MetaH	None	No	Comprehensive	Comprehensive	Partial
Rapide	Events modelled as Posets	Yes	Partial	None	None
Weaves	Structure conforms to Bipartite directed multigraphs	No	Comprehensive	Comprehensive	None
C2	None	Yes	Partial	None	None
Wright	Subset of CSP	Yes	Partial	None	None
ArchJava	Formal type system for architectural constructs only	Yes	Comprehensive	Comprehensive	Comprehensive
xADL 2.0	None	Yes	Partial	None	None
π-ADL.NET	π-calculus	Yes	Comprehensive	Comprehensive	Comprehensive

Table 2.2. Comparison of ADLs in Formalism, Architecture Style and SDLC Support.

2.2 Model Driven Architecture

The Model Driven Architecture [OMG01] or MDA is an approach to software design and development proposed by OMG. Looking at how this standard evolved, it can be seen as a logical organization of UML and related general or domain specific standards, formalizing an approach to describing software models and subsequently refining them to implementation. In MDA, software systems are defined iteratively in terms of computation independent, platform independent and platform specific models (CIM, PIM and PSM). These models allow a well organized partitioning of input from different participants in the SDLC while utilising the OMG modelling standards that fall under the MDA umbrella. The following advantages of the MDA approach are highlighted in [OMG01]:

- The clear demarcations at multiple levels of abstraction afforded by MDA allow users to replace lower level layers to benefit from technological evolution, while retaining the higher level models intact.
- The new detailed layers can benefit from partial automated generation based on information acquired from higher layers.
- The machine readability of MDA-based models makes visualization and maintenance much simpler.
- The information encoded in the models can be validated against specifications, can be tested against various infrastructures, and can be used to simulate system behaviour.

Note that for all of these benefits, comprehensive tool support is required. While OMG provides no implementation of its proposed standards, tool support is provided through third-party and open-source contribution [CT06]. Besides a variety of analysis and design related tasks, these tools also generate implementations based on PSM. Most of the tools available are applicable to OMG specifications in general, and do not apply specifically to MDA. The Eclipse Foundation [Ecl08] has developed implementations of many OMG specifications under the Eclipse Modelling Framework (EMF), Graphical Editing Framework (GEF) and the Graphical Modelling Framework (GMF). Together, these three frameworks form a modelling and code generation facility that automatically generates various components of support tools for a given model specification described in terms of a structured data model. Often Eclipse implementations do not conform strictly to OMG standards, offering instead approximations of OMG recommendations.
AndroMDA [And08] is another extensible generator framework based on MDA that transforms models into deployable components for multiple platforms. It offers a plug-in architecture that allows the development of *cartridges*, components that transform specific models into platform specific deployable components. A set of existing cartridges is already provided targeting some mainstream platforms such as Spring, EJB, Struts etc.

The π-ADL.NET approach differs from MDA since the standards under MDA provide a semi-formal notation. The syntax space for MDA is very large and thus difficult to be encompassed by any one tool. While the Executable UML (xUML) standard [MB02] for defining executable behaviour provides support for MDA, it is diagram based and lacks the formal base offered by π-ADL. In short, since MDA is an amalgam of notations and representations, it proposes the transition from model to implementation through a transformation in the medium of representation, which makes it difficult to analyze conformity between model and implementation, more so since the focus of MDA is forward engineering. In comparison, by using the architecture description as the foundation for implementation in π-ADL.NET, the implementation executable is guaranteed to remain faithful to the architectural model.

2.3 The B Formal Method

B [Sch01] is a formal method based on the Abstract Machine Notation. The development approach using the B method is to write an abstract model that represents a software system in terms of data, their properties, and services that process the data. This model is then refined accompanied by an automatic proof mechanism for systematic debugging to arrive at a concrete model, which can then be coded using an implementation language such as C++ or ADA. The development process guarantees that certain classes of error are eliminated, and thus unit and integration testing is not needed. Development under B proceeds with the following objectives:

- *To create correct software by construction*: It is well established that error correction is more cost effective in the earlier stages of the SDLC [Bro87]. The ability to detect and eliminate a large class of errors at the abstract model level allows cost savings.

- *To model systems in their environment*: The B method allows the modelling of the target system as well as its environment, including other systems, infrastructure, and human intervened operations.
- *To formalize specifications*: B specifications are formally founded on the Abstract Machine Notation.
- *To simplify programming*: By arriving stepwise at a concrete system model from an abstract model, the task of programming is highly simplified by eliminating a large amount of decisional effort from coding, and partially generating implementation code derived from the model.

Different tools are available for modelling and refining systems using the B method [Bme08]. Atelier B was designed to develop industrial B-Software projects. It consists of a set of tools integrated into a project management environment: a static checker, an automatic proof obligation generator, a set of automatic provers, an interactive prover, and code translators for ADA and C implementations.
Like B, π-ADL provides full support for formal description and development of software systems. Unlike B, π-ADL takes the architectural approach to software engineering. π-ADL.NET is set further apart by the fact that the model becomes the implementation. While the B process simplifies error free software development, it does not offer the capability to generate a complete implementation.

2.4 Conclusion

The survey presented in this chapter allows us to make a comparison of the implementation related objectives behind each technology or paradigm. Some of the projects reported come across as purely design solutions, providing mechanisms for modelling software architectures, which in turn act as high level blueprints for implementation. The ADLs Wright, C2, Rapide, Weaves and xADL 2.0 fall in that category. Other solutions provide support for modelling and refinement, and enable partial generation of implementation code. MetaH, MDA and the B Method fall into this latter category. The contribution of ArchJava is special in that it accomplishes an implementation that conforms to the architecture, while using the same language to architect and implement.
The π-ADL.NET project can be seen as an experiment in the logical opposite of ArchJava. While ArchJava leverages the existing syntax and tools of a development platform to implement support for architecture description, π-ADL.NET brings a formally founded ADL to an implementation platform. The research effort involved in π-ADL.NET is to primarily transform the ADL model to the implementation model, whereas ArchJava takes the implementation model and attempts to extend it to represent architectures. One advantage of π-ADL.NET is that a π-ADL based architecture remains formally verifiable at the implementation level, whereas only the ArchJava extensions to Java are formally verifiable, leaving the rest of the implementation without the benefits of rigorous formal analysis. Also π-ADL.NET successfully builds upon the formal design work that went into π-ADL, and brings the benefits of its architectural and formal foundations to the developer community.

Chapter 3: Background

This chapter provides the background study for the research presented in this dissertation. First the π-calculus is presented, with a focus on concepts pertinent to π-ADL (section 3.1). Secondly, an introduction to .NET technology and architecture is given insomuch as it pertains to π-ADL.NET (section 3.2).

3.1 The π-Calculus

The π-ADL is founded formally on the higher order typed π-calculus described in [Mil99][Mil93]. For the purpose of better identifying with π-ADL, I give a brief introduction to the π-calculus here. The π-calculus is a minimal Turing complete process calculus that provides constructs for specifying communicating processes that may change their configuration during the computation. It has found applications within and outside the domain of computer science, such as in cryptography, distributed systems and molecular biology [AG98][RSS01][Mil99]. A process P is defined recursively as being one of the following:

- $u(w).P$: Here u is a channel and w is a name. The construct $u(w)$ means that the channel u is waiting to receive w. This is known as an *input prefix*. Once the name w is received, the process P would execute. This is indicated by the . between $u(w)$ and P i.e. that P would execute *sequentially* after the input prefix $u(w)$ is done.
- $\bar{u}\langle w\rangle.P$: The notation $\bar{u}\langle w\rangle$ implies the inverse operation of $u(w)$, i.e. receiving the name w through the channel u (as opposed to sending). This is called an *output prefix*.
- $P|P$: The | symbol indicates parallel *composition*. This means that our process consists of two processes executing in parallel.
- $(\nu w)P$: The construct (νw) implies a new name w within the process P. All names are communication channels in π-calculus, and therefore a type specification is not needed in this construct. This construct restricts the name w within the context of P.
- $!P$: This term is used for *replication*, which means that the process P is composed with itself infinitely. See section 3.1.1 for formal definition.
- 0: This indicates the nil process, which has no remaining computations.

π-calculus processes have two types of names: free and bound. For any process, names being received through an input prefix, and names allocated through restriction, are bound. All the other names are free. The application of this concept is seen in the first rule of structural congruence below.
Abstractions in π-calculus are a means of reusing a process definition, with a provision for substituting a single name in the process instance. Instantiating a process from an abstraction is called a pseudo-application. For example if an abstraction F is defined as $(x).P$., then the pseudo-application Fv is an abbreviation for $P\{^v/_x\}$., which means that the name x is replaced by v in the process P.
As a modeling tool, π-calculus offers a set of structural congruence rules, and some reduction semantics. We look at each in turn.

3.1.1 Structural Congruence

The structural congruence rules of π-calculus are as follows:

i. $P \equiv Q$: This applies to two processes P and Q if Q can be obtained from P by renaming one or more bound names in P. This is known as *alpha-conversion*. For example the processes $u(w).\bar{u}\langle w\rangle$ and

$u(x).\bar{u}\langle x\rangle$ are alpha-inter-convertible (through the renaming of w to x or vice versa).

ii. $P|Q \equiv Q|P$, $(P|Q)|R \equiv R|(P|Q)$, and $P|0 \equiv P$. These axioms concern parallel composition, and essentially maintain that all possible compositions of a set of processes are equivalent to each other.

iii. $(vx)(vy)P \equiv (vy)(vx)P$ and $(vx)0 \equiv 0$: The order in which a name is created in a process is unimportant, and creating a name in a nil process results in a nil process.

iv. $!P \equiv P|!P$: This is the formal definition of replication.

v. $(vx)(P|Q) \equiv (vx)P|Q$: This rule applies only when x is not a free name of Q.

3.1.2 Reduction Semantics

A reduction is expressed in the form $P \rightarrow P'$. The following rules of reduction apply:

i. $\bar{u}\langle y\rangle.P|u(z).Q \rightarrow P|Q[z/y]$: This means that by renaming y as z, parallel input and output prefixes can be eliminated from the system.

ii. If $P \rightarrow Q$ then $P|R \rightarrow Q|R$: This means that a reduction preserved in the case of a parallel composition.

iii. If $P \rightarrow Q$ then $(vx)P \rightarrow (vx)Q$: A reduction is preserved in the case of a name creation.

iv. If $P \equiv P'$, $P' \rightarrow Q'$, $Q' \equiv Q$ then $P \rightarrow Q$: Processes that are structurally congruent have the same reductions.

3.1.3 Example

The following example illustrates the application of some of the rules governing π-calculus:

$\bar{u}\langle e\rangle.!(u(e)).0|u(o).\bar{o}\langle x\rangle.o(x).\bar{u}\langle o\rangle.0|e(z).\bar{e}\langle z\rangle.0$

In this example we have three parallel processes. Applying reduction rule i for the first two processes:

$!(u(e)).0|\bar{e}\langle x\rangle.e(x).\bar{u}\langle o\rangle.0|e(z).\bar{e}\langle z\rangle.0$

Note that o has been renamed to e for the second process, except for in the term $\bar{u}\langle o\rangle$. That is because in general the renaming does not extend into the inner scope of a prefix. Reducing the second and third process:

$!(u(e)).0|e(x).\bar{u}\langle o\rangle.0|\bar{e}\langle z\rangle.0$

Applying rule iv of structural congruence and reduction rule i:

$u(e).!(u(e)).0|\bar{u}\langle o\rangle.0|0$

$!(u(e)).0|0|0$

Applying rule ii of structural congruence:

$!(u(e)).0$

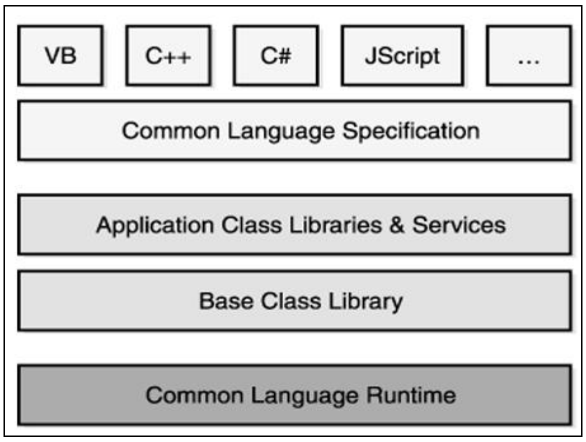
The subsequent section helps the reader map π-ADL on π-calculus as a high level representation of the formal concepts embodied by this method.

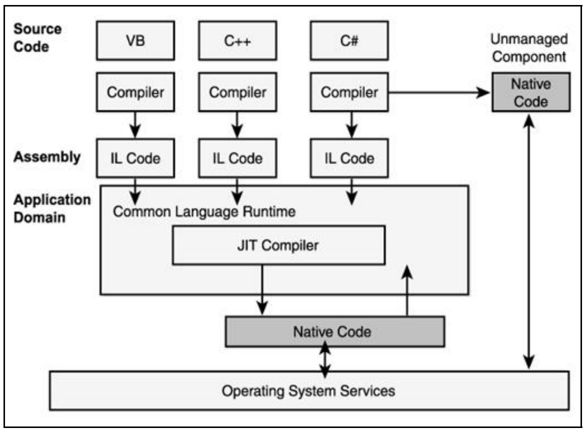
3.2 The .NET Framework

Microsoft .NET is one of the most popular technologies today for developing diverse types of software systems. Since before and after its release, a large number of books and articles have been written, explaining the design and merits of this technology. Here I will give a brief account of this technology as it pertains to the compiler development work for π-ADL.

3.2.1 The .NET Architecture

The design philosophy underlying the .NET technology, as described in [Lid02] and [Bur02] as well as numerous other texts, is to compile code into a platform independent binary of Common Intermediate Language or CIL. At the time of installation to the target machine or first execution, the CIL assembly is compiled *again* just in time to the native format. This gives .NET applications the advantages of being platform independent, as well as executing at the speed of native code. In fact it opens a new avenue of hardware specific optimizations for the Just In Time compiler, because it is compiling the executable on the same machine upon which the program will run.
Speed is scarcely the main claimed advantage of .NET though. Platform independence ensures that when developing for different target operating systems, the user need not worry about platform specific issues. In reality the .NET platform, as is developed and supported by Microsoft, is available only for different versions of Microsoft Windows operating systems. The Mono project [Mon08] implements the .NET virtual machine and runtime for the Linux, OSX and BSD platforms.
One of the main attractions of the .NET platform, from the development point of view, is that Microsoft .NET platform is accompanied by a very large library of reusable software components, in a diverse range of modern development and technology areas.
Further, by default the .NET applications execute with the Common Language Runtime (CLR). The CLR takes care of common programming tasks such as safe memory access, garbage collection, type safety etc. In essence, it provides all the services typically required by applications from the operating system, fulfilling it's commitment to platform independence.
The following figures 3.1 (a) and (b), borrowed from [Bur02], capture the overall architecture of Microsoft .NET, and the execution model of .NET applications respectively:


VB
C++
C#
JScript
...
Common Language Specification
Application Class Libraries & Services
Base Class Library
Common Language Runtime



(a) .NET Framework and the Common Language Runtime **(b) CLR Execution Model**

Figure 3.1

In figure 3.1 (a) we see the structural hierarchy of a .NET environment. Software developed using one or more .NET languages is compiled to the Common Intermediate Language i.e. CIL. This code accesses existing libraries provided by Microsoft or other sources, and executes under the supervision of the CLR, after being compiled to the native format. This is shown in figure 3.1 (b). Also, it is entirely possible to access existing software functionality from components developed outside the .NET framework. However while executing such components, it is not possible to use the services provided by CLR, and the security and memory management guarantees that come along with it.

3.2.2 Support for new .NET Languages

The .NET framework was not designed with any particular programming language in mind [Bur02]. Instead one of the goals of the .NET framework is to provide developers the flexibility of choosing the programming language of their liking while developing .NET applications. Many languages have been implemented for the .NET platform, and it is possible to implement others. The top-most layer in Figure 3.1 (a) has a box with ellipses, which is suggesting exactly that.
Developing a compiler for the .NET platform is like any other compiler project if we consider the CIL as an instruction set for a computer. CIL, in fact, is an instruction set for the CLR [Lid02] i.e. it conforms to the rules and regulations enforced by the CLR. The CIL packs a lot of features with syntax for namespaces, classes, methods, templates, events, exception handling, and string manipulation – in addition to what is normally found in assembly languages. An additional advantage is that a .NET compiler is a .NET application using all the advantages and features of the .NET platform, particularly the use of .NET components developed using any of the .NET languages. That makes the development effort a lot easier, and as I will show in Chapter 4, gives a lot of flexibility in defining the form the compiler's output will take.

Chapter 4: The π-ADL Compiler for .NET

This chapter describes in detail the π-ADL Compiler for the .NET platform, henceforth referred to as π-ADL.NET. In Section 4.1 I describe the overall architecture of the compiler. Section 4.2 describes the scanner implementation, which tokenizes the input code. Section 4.3 presents different components of the parser in order to describe the solution for transforming different aspects of π-ADL to CIL. Section 4.4 discusses the implementation of constructed data types, and the engineering decisions that went into integrating this complex area into the rest of the parser and the final executable assembly. Section 4.5 covers deviations in our implementation from the original π-ADL standard [Oqu05]. Section 4.6 concludes the chapter.
From the implementation perspective, the overall design of the compiler was guided by Edward G. Nigles' excellent book **Build Your Own .NET Language and Compiler** [Nig04]. The accompanying tools with this book were invaluable in modelling the main compiler algorithms.

4.1 The π-ADL.NET Compiler Architecture

The π-ADL.NET compiler is a multi-pass compiler implemented in C# for the .NET platform. Figure 4.1 shows the compiler object design. From an architectural point of view, the compilation of π-ADL code into CIL code is a two step process. The first step is the scan, in which the PiScanner object tokenizes the input code with meaningful classification of each token and returns a tokens array, with each token element encoding token value, type and position in the code file.

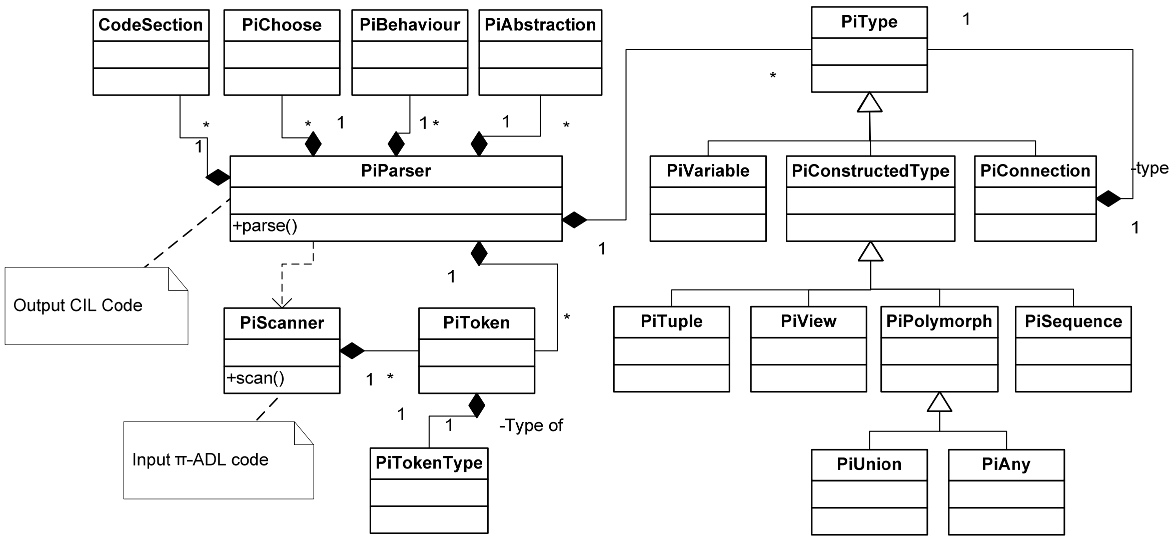


Figure 4.1. π-ADL.NET Compiler object model

As is evident from figure 4.1, a common current in the compiler object design is to provide a separate class for each language feature that warrants special treatment during compilation. The PiParser class contains the bulk of the functionality as it is mainly responsible for parsing each language construct (normally represented by an object of a corresponding class) as well as generating CIL code. Executables are created using the ilasm.exe tool provided along with the .NET SDK. Also, a π-ADL executable is dependant on a runtime library (more details in section 4.4), that was developed as part of this compiler design effort. Figure 4.2 illustrates the component architecture of the π-ADL.NET compiler and execution system architecture.

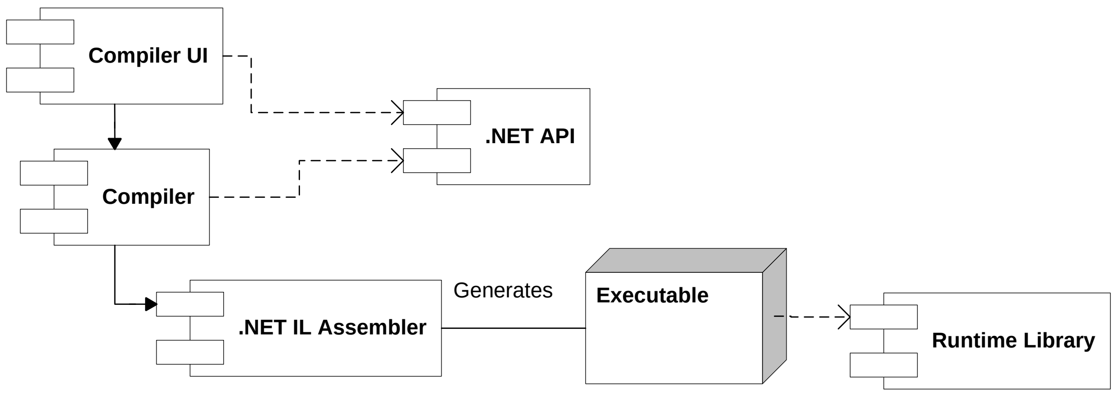


Figure 4.2. π-ADL.NET Component model

4.2 The Scanner

The π-ADL Scanner object, defined as the `PiScanner` object, is responsible for tokenizing the raw π-ADL code written by the programmer. It is based on the scanning algorithm described in [Nig04]. Listing 4.1 lists the pseudo-code for the scanning algorithm.

```
function scan(string source)
{
  int bestEndIndex = source.Length, currLine = 1, currIndex = 0;
  Array tokens;

  while (currIndex < source.Length)
  {
    bestEndIndex = source.Length;
    PiToken token = null;

    for each PiTokenType
    {
      Look for this PiTokenType in source starting from index = currIndex;
      if this is the leftmost and longest token found so far in the for loop
      {
        Set token to the value of this token, also setting the start and end
            indexes, token type, and line number equal to currLine;
        Set bestEndIndex to the end index of this token;
      }
    } //end for
    Add token to tokens;
    Skip white spaces;
    if (token.Type is Newline)
      increment currLine by 1;

    currIndex = bestEndIndex + 1;
  } //end while
} //end scan
```

Listing 4.1. π-ADL.NET Scanner Algorithm Pseudo-code

In reviewing the pseudo-code in listing 4.1, we see that the scanner algorithm starts from the position `currIndex`, which is initially 0, of the `source` and attempts to find the first occurrence of each token type. For amongst those token types that are found, it selects the *leftmost* and *longest* and adds it to the array `tokens`. `currIndex` is then incremented to point to the character after the end index of the token found, and the search is started again.
The selection criteria of leftmost and longest demand some explanation. Leftmost makes sense because we are looking for the first next token. The criteria longest is needed because it is entirely possible to have a candidate token residing inside another token. For example consider the string

`"+sample"`. This is a string token that contains a candidate operator token i.e. the + sign. However it is clear that attempting to parse the operator will not only be incorrect but will also lead to an error when parsing the remaining part `sample"`. The criteria of leftmost *and* longest therefore compensate for that and parse the complete string correctly.

```
public class PiTokenType
    {
        public const int tokenCount = 13;

        public enum Types
        {
            Identifier      = 1,       //good for identifiers,
                                       //keywords and boolean values
            UnSignedFloat   = 2,
            UnSignedInteger = 3,
            String          = 4,
            Comment         = 5, // //
            Operator        = 6,
            SquareBracket   = 7, // [
            CurlyBracket    = 8, // {
            Parenthesis     = 9, // (
            SemiColon       = 10,
            Comma           = 11,
            Colon           = 12,
            Newline         = 13,
            Error           = 14
        }
    }
```

Listing 4.2. The PiTokenType Class

Besides implementing the core algorithm, the scanner class fires a scanErrorEvent every time an invalid token is encountered. Along with the event a PiToken object is emitted that indicates the value and position of the invalid token. After firing the error event, the scanner continues to parse the rest of the code.
One important implementation specific issue of optimisation was found to be associated with finding the position of a token in the code text. While trying to find the leftmost occurrence of a candidate token, initially the following overload of the String.IndexOf method was used:

```
String.IndexOf(String value, Int32 index);
```

The argument `value` specifies the sub-string to be searched, which in our case would be one of the tokens. The argument `index` specifies the position in the source string onwards from which the word is to be searched. This method performs a case and culture sensitive search for the string provided, using the current culture. For one of the test programs spanning 500 lines of code, the scanner implementation using this method overload took 50 seconds to complete the scan. In order to reduce the scan time, the following overload was used:

```
String.IndexOf(String value, Int32 index, StringComparison comparisonType);
```

This overload takes a third enumeration argument which allows the programmer to specify one of a list of comparison options. Choices include using the current culture or the invariant culture, case sensitivity, or the use of ordinal sort rules. This method was employed with the ordinal option which enforces string comparisons simply based on the character numeric values. Using this comparatively simple search criterion the scan time for the same program was reduced from 50 seconds to 6 seconds.
The code listing of the class PiTokenType in Listing 4.2 shows the types of tokens the PiScanner is capable of identifying.

4.3 The Parser

The parser algorithm, encoded in the `PiParser` class, is a top-down recursive descent predictive algorithm, as described in [Wik07]. It operates on the list of tokens output by the scanner, and has the following features:

- A top level routine initiates the parse by imposing a structure on the tokens list, in the form of one or more non-terminal or terminal symbols. In our case and in most others cases, this structure corresponds to the top level BNF production.
- In order to verify that the structure of the code corresponds to that imposed by the top level BNF production (`program := explicitBehaviourDeclaration+` in our case), each one of the symbols in the definition are parsed. If a symbol is terminal then the method returns true or false immediately, depending upon whether the parse was successful or not. If it is non-terminal then a parse is performed on the same lines as in the case of the top level production.
- The parse tree corresponds to the BNF specification presented in Appendix II. Where considered appropriate, CIL code is generated by firing a parse event. In this way, as the parse proceeds, the code is generated incrementally up till the point when the main parsing routine terminates. In case of a successful parse, the Boolean value true is returned. In case of failure, false is returned.

When generating code, a large performance issue emerged with the use of ordinary string concatenation. That is because a C# string, which corresponds directly to a CIL string primitive, is physically allocated as an array of constant length. Concatenating two strings implies allocating a new string and copying the contents of the two strings into it. To optimise that, during the course of execution leading to a code emitting event, a StringBuffer object is now used. Another implemented optimization is to avoid code generation altogether in some cases by externalizing the executable functionality to runtime libraries (more on this later).
Since compiler design theory – as is used in this research – is well documented in literature, perhaps the main challenge in designing the compiler i.e. the un-chartered territory in the project domain, was to create a mapping between π-ADL and CIL. CIL is a stack based language that defines the instruction set of the Common Language Runtime [Lid02]. π-ADL is a formally founded language based on the π-calculus, defining a **behaviour** or an **abstraction** as its atomic functional unit. This created some interesting peculiarities in the compiler implementation. For example:

- Behaviors and abstractions communicate through connections, which have no counterpart in CIL or the .NET framework. Thus a .NET class was developed to emulate π-ADL connections, with the requisite threading and synchronization functionality.
- Behaviors and abstractions are the units of execution themselves, and also encapsulate the connection and variable declarations. This gives the functionality of both classes and methods in CIL. Behaviour declarations thus correspond to a class plus an entry point method declaration in CIL.
- In π-ADL, behaviors can contain sub-blocks of code that can run concurrently, when using choose and compose blocks. There is no possibility of declaring a method inside a method in CIL, and this leads to deferring the behavioral code generation for choose and compose blocks till the end of the parse. The behaviors corresponding to the choose and compose blocks are implemented as separate methods outside the entry point method, and are called in an appropriate manner from the method corresponding to their containing behaviour.
- π-ADL constructed data types of tuple, view, union, any and sequence can be neither mapped to .NET primitives, nor directly to any of the meta types available in .NET, such as classes. Specialized class implementations for each π-ADL constructed data type were therefore needed to cover the exact functionality expected from them.

In the rest of this section, the code generation issues and their resolutions are described using sample π-ADL code and its corresponding CIL code or C# code, where the functionality resides in runtime libraries.

4.3.1 Compiling π-ADL Parallel Processing Constructs

As mentioned earlier, each behaviour definition in a π-ADL program results in a separate thread of execution at startup. Other cases where parallel processing occurs are the compose, choose and replicate blocks. Here we treat the runtime implementation of each of these π-ADL constructs in turn.

4.3.1.1 Behaviour implementation

A π-ADL behaviour is compiled as a separate class in CIL. Variables and Connections declared in a π-ADL behaviour become class variables in CIL. While the π-ADL assumes its variables and connections to be initialized with declaration, it has to be done explicitly in CIL inside the default constructor for the behaviour class. The CIL code corresponding to the functionality defined in the π-ADL code for the behaviour is generated in the ep$ method, which is effectively the entry point of execution for each behaviour class. Note that the CIL representation of an abstraction also has a similar ep$ method, and is called against a pseudo-application in π-ADL. The entry point of a compiled π-ADL.NET program is the main method of an internal class Controller$. After all the behaviours are identified, the main method of the Controller$ class is generated.
The function of the Controller$ main method is to simply instantiate each behaviour class, and for each instance create a corresponding thread object. Each thread is then started with the ep$ method of the behaviour as its startup method. The main method of the Controller$ class initiates each behaviour thread in the order in which the behaviour is declared in the source program. However the actual execution order is determined by the .NET runtime.

4.3.1.2 Implementing compose

Semantically, π-ADL requires the sub-blocks of a compose block to execute in parallel without any precedence requirements. In order to accomplish that in CIL, the logic of the sub blocks is output in separate methods for each sub-block. These methods belong to the class representing the behaviour or abstraction and are called from the ep$ method of that behaviour or abstraction. The methods are named using the notation "method<methodIndex>", where methodIndex is a global integral value incremented each time a compose or choose sub-block is encountered.
To implement parallelism for all the sub-blocks inside compose, a separate thread is created for each sub-block and executes the method generated for that sub-block. If there is a compose block nested inside another compose block, more methods are created at the class level; there is no nesting of methods comparable to the representation in π-ADL, since such a nesting is not supported by CIL. Instead, the methods are named using the above mentioned convention and simply called from the method representing the parent sub-block, using the same threaded approach as the one used for the top level compose block. The π-ADL.NET compiler does not impose any limit on the number of nesting levels for any of the compose, choose or replicate constructs, and any of these constructs can be nested within each other in an arbitrary order.

4.3.1.3 Implementing choose

The logic of each of the different sub-blocks inside the choose block is output inside a separate method, very much like it is done for compose blocks. Each of these methods is invoked through a separate thread, as in compose. However the threads for the choose sub-blocks are in competition and the successful thread needs to be able to terminate the rest before it can execute. A basic approach would be that the first thread to execute its first statement would terminate the others. However there is a possibility that while one thread is busy terminating the other threads, another thread may resume execution and start the termination routine on its own. Therefore in addition to the task of terminating all competing threads, the chosen thread should also be able to signal the other threads of its nomination.
Figure 4.3 shows the data structures maintained by the π-ADL.NET runtime for thread management

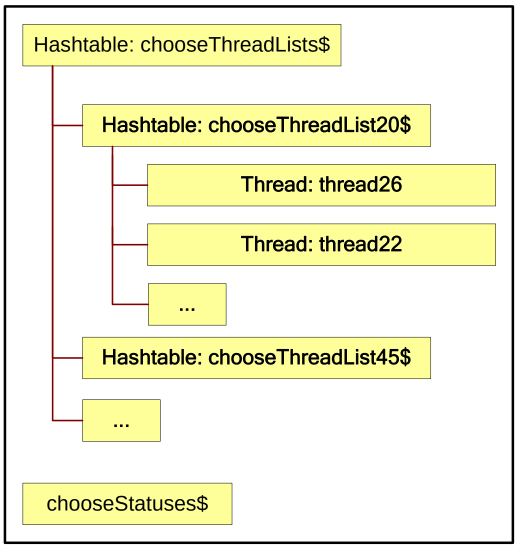

Hashtable: chooseThreadLists$
Hashtable: chooseThreadList20$
Thread: thread26
Thread: thread22
...
Hashtable: chooseThreadList45$
...
chooseStatuses$

Figure 4.3. Runtime metadata maintained by a π-ADL.NET executable for managing Choose constructs variables in PiParser

and signaling. Each choose block has a corresponding hash table named chooseThreadList<x>$, where the value of x is the integral index assigned to the choose keyword by the scanner. A global hash table chooseThreadLists$ maintains reference to all the chooseThreadList<x>$ objects.

Each of these hashtables in turn maintains a reference to all threads corresponding to the sub-blocks of its associated choose block. In order to support signaling, the hashtable chooseStatuses$ is used. It contains reference to each choose block by its token index number, and against this key it maintains a Boolean value indicating whether one of its threads has commenced execution. By default this value is false for each choose block. Upon starting execution, any choose sub-block thread does the following:

a) Acquire a lock on chooseStatuses$ by using the Monitor.Enter method.
b) Check Boolean value for the associated choose block in chooseStatuses$. If true then branch to e).
c) Update the value for the associated choose block in chooseStatuses$ to true.
d) Call a special internal method cleanupChooseNonTermins$ to terminate all the threads for the given choose block. This method takes a string argument for the name of the chosen thread so as to exlude it from the termination sequence.
e) Release the lock on chooseStatuses$ by using the Monitor.Exit method.
f) If the value examined in b) is true then the method returns and the thread terminates. Otherwise it continues executing the logic encoded in its associated sub-block.

Special case for abstractions:This algorithm ensures that one and only one thread is executed for any choose block. However there is one caveat in the metadata structure which becomes evident when multiple instances of the same abstraction are executing in parallel. Since the chooseThreadList<x>$ variables attempt to maintain exclusivity of executing choose blocks through their unique indexed position in the π-ADL code, that property is not applicable when the same code is executing in different threads at the same time.

To resolve this issue, a custom class Int64Obj has been developed to encapsulate a 64-bit integral value. Each class representing an abstraction contains a static variable of this type named choose$exe$number. In addition, a local variable choose$exe$local of type Int64Obj is instantiated for each choose block found at some level in an abstraction. Every time the abstraction is executed, choose$exe$number is incremented and this local variable is assigned the resulting value in a thread safe manner so that each abstraction instance holds a unique value for it's choose$exe$local.

Subsequently, the entry made to the chooseStatuses$ hash table incorporates the value of the thread unique choose$exe$local variable in its key value, adding the desired level of differentiation in naming to allow multiple abstractions to be identified separately. Furthermore, in the case of abstractions the naming format of the hash table chooseThreadList<x>$ is modified to chooseThreadList<x>$-<choose$exe$local> to achieve the desired level of differentiation. This allows cleanupChooseNonTermins$ to terminate only those threads that belong to the one particular abstraction instance being executed.

For the above modification to accommodate abstractions, we observe that modifying the design thus has the desired effect of exclusivity without changing the above mentioned algorithm, or drastically modifying the structure in figure 4.3.

4.3.1.4 Implementing replicate

Semantically, the grammar of the π-ADL replicate construct can be recursively defined as:

```
replicate{<set of statements>} ::= compose{<set of statements> and replicate{<set of statements>}}
```

```
behaviour {
  compose {
    via myAbs send 42;
  and
    via myAbs send 42;
  and
    via myAbs send 42;
  }
}
value myAbs is abstraction (argi : Integer)
{
  i : Integer;
  replicate {
    via in receive i;
    i = i * argi;
   via out send i;
  }
}
```

Listing 4.3 Simulating controlled concurrent replication

This implies that an infinite number of threads will be created to execute the <set of statements> in parallel. While this syntax establishes a close correspondence between the replication concepts of π-ADL and π-Calculus, it creates obvious problems for the compiler implementation, where an arbitrary number of parallel processes will quickly overrun the processor and memory resources of the system executing a π-ADL program. To resolve this problem, the following solutions were considered:

a) Have a preset maximum number of parallel threads that can execute at any given time. As soon as one thread terminates, another one is launched.
b) Extend the replicate syntax to allow the provision of an integral parameter. The value of this parameter will specify to the system the maximum number of parallel threads to be launched at any given time. Compared to a), this approach gives the flexibility of tuning the concurrency level according to the problem at hand.
c) Have only one thread active at any given time. This would give the computational equivalent of an infinite iterative loop.

The π-ADL.NET compiler implements choice c). The rationale is that although the approach of a) allows a larger problem set to be executed in conformance with the π-calculus formalism, no preset value for the maximum number of parallel threads can ensure the π-calculus conformant execution of all possible programs. Choice b) at least gives problem specific assurances for the correct simulation of the replicate formalism, but at the cost of deviating from π-ADL syntax. The advantage of c) is the simplicity of the implementation while conforming to the π-ADL syntax. Using c), a functionality similiar to b) can be implemented through other means, such as illustrated in Listing 4.3.

4.3.2 Compiling π-ADL Connections

In developing the π-ADL.NET compiler, the implementation of connection related functionality went through multiple iterations before reaching its present form. The reasons for this are the following issues:

- The send and receive operations have to be atomic. The possibility of modifying the variable being transferred in a parallel thread should be avoided.
- The Connections must hold a provision for having multiple send operations queued up. Receive operations must execute without the loss of any data sent in such a case.
- Connections must also cater to the synchronization needs of choose statements in-case the first statement of a choose sub-block is a receive statement.

- Connections are mobile. They can be sent and received through other connections, and can be passed as arguments to abstractions, and are therefore full-fledged data types in π-ADL in line with the polyadic π-calculus. Earlier compiler implementations created a new CIL connection class against each π-ADL connection declaration, and then instantiated it. However, mobility entails that both the sender and recipient share the same class definition. To retain both definitional consistency and type flexibility, a single generic connection class is defined.

Listing 4.4 shows the Connection class implementation in C#. The CIL generated and used by the π-ADL.NET executable is isomorphic with this code. Looking at the class declaration in line 1 we see that the class is declared as a .NET generic class, similar to a C++ template class. The variable declarations on lines 2, 3 and 4 assist in the correct functionality of the send and receive methods. There are 2 AutoResetEvent instances declared in line 2.
An AutoResetEvent is like a logical gate. Threads block on an AutoResetEvent object when they call its WaitOne() method, and if the AutoResetEvent object is in the non-signaled state. When the Set() method is called on the AutoResetEvent object, the waiting thread is unblocked and resumes execution.
The AutoResetEvent object allows only one thread to be unblocked for each Set() call, and reverts to the non-signaled state thereafter [MSD08].

```
1.  public class ConnectionTemplate<T> :
    AbstractConnection {
2.  AutoResetEvent e, receiveE;
3.  long sendWaiting, receiveWaiting;
4.  Queue<T> valQueue;
5.  private string toStringVal;
6.  public ConnectionTemplate() {
7.  e = new AutoResetEvent(false);
8.  receiveE = new AutoResetEvent(false);
9.  sendWaiting = 0;
10. receiveWaiting = 0;
11. valQueue = new Queue<T>();
12. }
13. public void send(T a)
14. {
15. bool noException = true;
16. valQueue.Enqueue(a);
17. do {
18. if(Interlocked.Read(ref receiveWaiting)==0)
19. {
20. Interlocked.Increment(ref sendWaiting);
21. e.WaitOne();
22. Interlocked.Decrement(ref sendWaiting);
23. }
24. else {
25. try {
26. e.Set();
27. receiveE.WaitOne();
28. }
29. catch (System.InvalidOperationException)
30. {
31. noException = false;
32. }
33. }
34. } while (noException == false);
35. }
36.
37. public void receive(ref T a)
38. {
39. bool noException = true;
40. do {
41. if (Interlocked.Read(ref sendWaiting) == 0)
    {
42. Interlocked.Increment(ref receiveWaiting);
43. e.WaitOne();
44. a = valQueue.Dequeue();
45. Interlocked.Decrement(ref receiveWaiting);
46. receiveE.Set();
47. }
48. else {
49. try {
50. a = valQueue.Dequeue();
51. e.Set();
52. noException = true;
53. }
54. catch (System.InvalidOperationException) {
55. noException = false;
56. }
57. }
58. } while (noException == false);
59. }

60. public void receive(ref T a, ref Hashtable
    chooseStatuses) {
61. bool noException = true;
62. do {
63. if (Interlocked.Read(ref sendWaiting) == 0) {
64. Interlocked.Increment(ref receiveWaiting);
65. e.WaitOne();
66. Monitor.Enter(chooseStatuses);
67. a = valQueue.Dequeue();
68. Interlocked.Decrement(ref receiveWaiting);
69. receiveE.Set();
70. }
71. else {
72. Monitor.Enter(chooseStatuses);
73. try {
74. a = valQueue.Dequeue();
75. e.Set();
76. noException = true;
77. }
78. catch (System.InvalidOperationException) {
79. noException = false;
80. }
81. }
82. } while (noException == false);
83. }

84. public bool IsEquivalent(object x) { /*…*/}
85. public void setToString(string val) { /*…*/}
86. public override string ToString(){ /*…*/}
87. public void deepClone(out object obj) { /*…*/}
88. }
```

Listing 4.4 Connection implementation for π-ADL.NET (C# code)

The Interlocked class used both in the send and receive methods provides atomic operations for variables that are shared by multiple threads [MSD08a]. The two methods Increment and Decrement of the Interlocked class used in the send and receive methods are used to atomically increment and decrement long values. The Read method reads the value of a long variable atomically.
As can be seen in the implementation in Listing 4.4, AutoResetEvents, the Interlocked class and the Monitor class are employed to ensure correct synchronous interaction between the send and receive methods.
The synchronous communication facility provided by the send and receive methods can result in a myriad different contexts of interaction. This can occasionally result in unexpected forms of interaction, resulting in the .NET framework level exceptions. A simple and practically proven solution is to re-try the send or receive operation, as can be seen in the exception handling code in lines 29, 54 and 78.
We also note that there are two versions of the receive method. The one declared at line 60 is specifically written for choose blocks to ensure thread safe execution of the choose algorithm explained in Section 4.3.1.3 as well as of the receive operation.

4.4 Constructed Data Types

The π-ADL.NET constructed data types are implemented as distinct classes in an external C# library linked with executable π-ADL code at runtime. This helps keep the executable size small, with a minor performance penalty. Also this approach helped to simplify the implementation of data type functionality, which would otherwise have been a difficult and time consuming exercise in CIL programming.
While the existing variety of constructed data types give the π-ADL.NET implementation a rich syntactic basis for modeling the structure and behavior of a software architecture, the implementation experience shows that a more viable approach would have been to provide a meta-type system equivalent to the classes and objects in object oriented languages. Using such a framework would enable the user to easily recreate all the properties embodied by the specialized data types and more, with relatively little complexity of implementation at the compiler level. This topic is discussed in detail in the next chapter.
The formal specification work required to develop a meta-type system compatible with the π-calculus foundations of π-ADL is well outside the domain of the π-ADL.NET project, and can form a line of separate research work in the future.
The current implementation of constructed types does not support type definition i.e. the definition of a particular connection, tuple, view, union, any or sequence global signature that can be used in behaviours and abstractions. Without this feature, the programmer has to define and redefine a particular constructed type each time he intends to use it, causing programming overhead. The implementation of type definition functionality has also been identified as future work, although by providing a meta-type system implementation we may alleviate that need.
Typically each one of the implemented data types support a standard set of methods that assure first class citizenship in the language as well as easy interaction with the environment. This includes declaration and initialization, assignment, I/O through normal connections as well as the special `in` and `out` connections for console interaction, and sending as argument in a pseudo-application. In addition there are type specific functions. For example tuples and views support projections, union and any types have a type-select-case structure applied to them and so on. Not only the ability to perform these type specific operations has been implemented, but the functionality to process other types in these peculiar operations has also been provided. Appendix II lists code for selected runtime classes representing or helping some of the constructed data types, namely view, union and sequence. The relationship shown in figure 4.1 between the compiler classes corresponding to the runtime classes applies exactly the same way between the classes in Appendix II, which have been written in C#. Here we briefly discuss each one of these classes in turn. Some methods have common functionality across all data types and are thus discussed only within the context of one data type.

4.4.1 The PiConstructedType class

This class is the base class for all constructed data types. It has an empty default constructor, and implements the following methods, all of them public:

```
public bool isConstructedType(string type)
public string NetToPiADLTypeName(string type)
public virtual bool deepCompare(PiConstructedType t)
```

The `isConstructedType` method is used by `PiConstructedType` and its descendant classes to determine whether the value of the string type is the name of a π-ADL constructed data type or not. This is useful in multiple different contexts. It returns true for all constructed types, except for Connnections, which are treated differently. `NetToPiADLTypeName` takes a string parameter containing the name of a CIL primitive type, or the .NET class name of one of the constructed types. The function returns the π-ADL name for the type after evaluating its value using a switch-case statement block. For example, if you pass the string "float64", this function will return "Float", since in CIL a π-ADL Float is represented by the data type float64.
The method `deepCompare` is a virtual method, and therefore has no implementation. This makes sure that all classes extending `PiConstructedType` implement this method, and this method can be invoked on those classes even if they are upcasted to `PiConstructedType`. The purpose of this method in all concrete implementations is to find out whether the host object has the same type as that of the argument object, and where applicable, they also have the same type structure.

4.4.2 The View class

The `View` class has three public fields of type `ArrayList`: `TypeData`, `Values` and `Names`. It has a default constructor that initializes the fields, and implements the following public methods:

```
public int addType(string type, string name, object val)
public object getValue(int index)
public object getValue(string name)
public void getValueRef(string name, out object obj)
public bool setValue(string name, object obj)
public bool setValue(string name, object obj, string polymorphType)
public bool setValue(int index, object obj)
public override string ToString()
public void deepCopy(View v)
public void deepClone(View v)
public override bool deepCompare(PiConstructedType ct)
public bool fillViewFromConsole(String prefix)
```

The `addType` method accepts the type, name and value of a data element. This information is then added to the appropriate array lists, where a given index value defines a coherent record across the different array lists. The method `getValue` comes with two different variations, allowing retrieval of an element from the `Values` array list by its name or its index. The `getValueRef` method redirects the reference of the argument `obj` to point to the element referred to by the string `name`. Similarly to `getValue`, the `setValue` method has two overrides that allow updates by index and by name respectively. A third override takes an extra argument defining the type of the second argument `obj`. This is needed for updating union and any type elements in a view. There is good reason to update a union, instead of replacing it. Replacing a union completely would entail unnecessarily rebuilding and replacing the list of all possible types associated with it. An update of the current type and value avoids that overhead. Note that the same reason does not apply to the any type, but it is updated using the same method as a union simply because of its relative similarity with a union (this is the concept of logical cohesion at work).

The `ToString` method creates a human readable string based on the elements and their contents within a view. The same functionality is to be expected from this override for other data types, and examples can be found in Tables 5.4 and 5.5 (next chapter). The `deepCopy` method is responsible for deep copying a view to another, and is used when assigning one view value to another in π-ADL. It works by processing each member of the view in turn. For basic types such as String, Integer etc., it would simply copy the value from the source view to the target view. For constructed types, the `deepCopy` method of that constructed type is invoked in order to make a copy of that object and save it within the appropriate view. This implementation of `deepCopy` for the view class as well as that of other data types is recursive, and therefore in all cases it is defined in terms of the `deepCopy` methods of other constructed data types. The method `deepClone` is similar in structure to `deepCopy` and is functionally different in that it creates a clone of each member of the argument view, and adds it to the list of the elements of the method host view. This method is employed when a view is being sent via a connection. The `deepClone` method of other constructed types has the same specification and application.
The implementation of `deepCompare` is also recursive. It first checks whether the argument is also of the type View. Next, it establishes whether both the host and the argument have the same number of elements. Finally it compares each one of the elements. For basic types, a simple type comparison suffices. For, constructed types, their respective `deepCompare` methods are invoked. Finally for connections, their `IsEquivalent` method is invoked, the implementation of which is not shown in Listing 4.4. It is coded as follows.

```
public bool IsEquivalent(object x)
{
    Type t = x.GetType();
    Type tt = this.GetType();

    if (t.ToString() == tt.ToString())
          return true;

    return false;
}
```

This method needs to be used since the Connection class is a template. However, the `ToString` method of a `System.Type` object representing a template also encodes the selected type of the template in the returned value, thus making it unique for every different type assigned to a template. The above usage is not documented in the .NET class reference [MSD07], but our improvisation has functioned correctly in all tests conducted so far.
Finally, the `fillViewFromConsole` method provides a managed interface for the user to easily enter element values into a view. It does so by prompting the user with the name and type of the element to be entered. If one or more elements of a view are constructed data types, then the adequate `fill<Type>FromConsole` method is called. Adequately named console input methods exist for all constructed data types, e.g. `fillUnionFromConsole`, `fillSequenceFromConsole` etc. Like other methods employed by constructed data types, this method is also recursive.

4.4.3 The Polymorph class

Being the base class for Union and Any, this class provides the data types and a common method used by both of these classes. The data types are:

```
public object CurrentValue;
public string CurrentType;
```

The string `CurrentType` contains the .NET name for the current type of the `Polymorph`. The object `CurrentValue` refers to an object of the constructed type itself, or in case of primitive types, their boxed value. In addition, `Polymorph` implements the following method:

```
public bool isTypeEqual(Polymorph p)
```

This method is responsible for determining the equality of the current type of the host and the argument `Polymorph` objects. If the current type is a constructed type, then the `deepCompare` method is invoked. For connections, the `isEquivalent` method is invoked in order to compare the two objects.

4.4.4 The Union class

In addition to the fields provided by `Polymorph`, the `Union` class has the following two fields:

```
public ArrayList TypeData;
public ArrayList TypeStructures;
```

`TypeData` contains the names of the possible data types allowed in the union. `TypeStructures` contains the structure of the possible constructed types allowed in this union. They assist in creating new values based on their structures when the `CurrentValue` and `CurrentType` fields are updated. The value of an entry in `TypeStructures` is only relevant for constructed data types, and for a basic data type it assumes the value of an empty string.
In section 4.4.2, all the common methods have been discussed, so there are only a few remaining methods that are unique for the `Union` class. However there are slight contextual variations that I will discuss briefly. Besides the default initialization constructor, the methods are:

```
public int addType(string type, object structure)
public bool setValue(string type, object val)
public bool setValue(object val)
public override bool deepCompare(PiConstructedType ct)
public void deepCopy(Union u)
public void deepClone(Union u)
public override string ToString()
public bool fillUnionFromConsole(String prefix)
```

The method `addType` adds a type name to the `TypeData` array list, and a type structure to `TypeStructures`. It is used when a union is declared in π-ADL, or when `deepClone` is invoked. The method `setValue` has two overloads. The one that takes two arguments is invoked when both the current type and the current value of the union is changed. The second overload is used when only the value is being updated, and there is no change in the current type. In `deepCompare`, the possible type lists of the two unions are compared. The current type or its value does not factor in the evaluation. `deepCopy` copies the current type and its value from the argument union to the host union object. `deepClone` does that, as well as copies the list of possible types by duplicating the array lists `TypeData` and `TypeStructures`. `ToString` generates a human readable string, and `fillUnionFromConsole` prompts the user to select the current type from the list of possible types, and then enter the value, as discussed in the previous section.
Note that there is no need for extra support methods for use of a union in a type select-case block, or in using the type equality operator. The `deepCompare` method is used in both cases.

4.4.5 The Sequence class

The class `Sequence` defines the following three fields of its own:

```
public ArrayList Values;
public string Type;
public object Structure;
```

The string `Type` contains the name of the data type for which this sequence is defined. The object `Structure` contains an object representing the type if it is a constructed type, or an empty string otherwise. The `ArrayList Values` contain the values stored in the sequence. The following methods are defined in the class `Sequence`:

```
public Sequence()
public void setType(string _type, object _structure)
public void addValue(object val)
public void setValue(object val, long index)
public void setValue(string type, object val, long index) //for union and any
public object getValue(long index)
public override string ToString()
public void deepCopy(Sequence s)
public void deepClone(Sequence s)
public bool fillSequenceFromConsole(String prefix)
public override bool deepCompare(PiConstructedType ct)
```

The method `setType` simply sets the values of the fields `Type` and `Structure` to those of the arguments. `addValue` inserts the argument `val` to the end of the array list `Values`. `setValue` has two overloads. The one with two arguments updates the value at a particular index. The other has been developed specifically for unions and any data types i.e. when you have a sequence of type union or any. It follows the same rationale as the method `setValue` for the `View` class. Both versions of `setValue` have an inherent flexibility in that if the provided index value is greater than the last populated index of the array list, then the array list is extended to accommodate that, and the intermediate positions are filled with deep clones of the field `Structure`. `getValue` returns the element at the specified index. `deepCopy` makes duplicates of all elements in `Values`. `deepClone` clones the field `Structure`, and copies the string `Type` to the target sequence, in addition to invoking `deepCopy` in order to duplicate `Values`. The method `fillSequenceFromConsole` allows the user to specify the number of fields he intends to input, and then facilitates the input of those values.

4.4.6 Analysis

The fundamental design pattern in the π-ADL.NET data type runtimes is the cross-reliance of the different types to accomplish different operations. This has the effect of keeping the code base reasonably small and easy to understand. Further, the advantage of implementing the data types in the runtime libraries becomes evident where explicit operations in CIL become implicit in C#. For example, assigning and accessing basic types to the object references (e.g. `CurrentType` in `Union`, `Values` in `View` etc.) would have required the boxing and un-boxing of those values in CIL. In C#, this is handled automatically without the programmer's intervention.
Finally, special attention has been paid to support intuitive and error free input from the console. The constrained input system assures that errors are processed early on in the input pipeline, avoiding the need for extensive error checking in other code components. The approach is vindicated by the principle of end-to-end complexity in system design [HS02].

4.5 Syntax Deviations

The π-ADL.NET compiler currently compiles most of π-ADL, barring some features pertinent to describing non-deterministic behaviour. The implementation of the set, bag, location and quote constructed data types has been left out, as well as the iterate looping construct. The set and bag types are similar. A set is an unordered collection of elements, which doesn't support duplicates. A bag is like a set except that it supports duplicates. A location is a typed data store. A quote defines a distinct type by pairing a quote type and a name, and performs the same function as an enumeration.
The iterate construct loops over a set or a bag in a non-deterministic fashion, so that different executions will result in a different order of retrieval of the constituent elements. π-ADL.NET currently implements a C-style while loop instead.
The above set of features can be implemented in π-ADL.NET at a later stage. The reason why they were left out in the current phase is due to the focus on implementation specific features of the language.

The currently implemented π-ADL subset is Turing complete and provides coverage for all the operations derived from π-calculus. In subsequent development phases, it may implement the remaining π-ADL specification and will add implementation specific extensions to the language syntax for providing access to external .NET class libraries. These extensions have already been designed and are described in Chapter 6.

4.6 Chapter Conclusion

The π-ADL.NET compiler constitutes the tool support necessary to experiment with the use of π-ADL as an implementation language.
Apart from the core compiler design work, I have done some prototyping work to use the compiler as a foundation for a visual programming and architecture design environment using the GME tool, and in exploring approaches to the use of 3D modeling and animation in defining a viable visual syntax. This line of work is discussed in [QDO06] along with some initial results.
Under current development is a DirectX based 3D programming and modeling interface being developed as a plug-in to Microsoft Visual Studio 2005 and later. The objective of this initiative is provide a 3D visual interface to the π-ADL.NET compiler and experiment with concepts such as program simulation via 3D animation, visual debugging, simultaneous multi-aspect views of the architecture during design and simulated execution, and the simultaneous orchestration of these concepts. This project will form the core of the next generation π-ADL toolset as a complement to the π-ADL.NET compiler.
About 1500 lines of code have been written to test the various functions of the π-ADL.NET compiler. In addition, a functional web-service architecture has been developed in collaboration with the computer science department of the University of Rey Juan Carlos, constituting a novel application of π-ADL compared to pre-π-ADL.NET modeling work. This is discussed in Chapter 6. The π-ADL.NET compiler has been used at the VALORIA laboratory for modeling multi-agent systems, as well as the High Level Architecture (HLA) [IEE00]. The parallel and high-level nature of π-ADL makes it suitable for development work in all of these application areas. There is also some current work at our laboratory in visually modeling π-ADL and generating code using the DSL tools for Visual Studio .NET. The DSL tools are reported in [GS03].
The long term objective of the π-ADL.NET project is to test the modeling of more and more application domains, and consequently improve the toolset towards an industry strength solution for software architecture modeling.

4.6.1 Related work

While there is no existing research work in compiling a π-calculus based language or an ADL to the .NET platform, there are some compilers for formally founded languages for .NET. One notable example is the .NET compiler for the F# language [SM07], which is based on the ML language [MTR97], a formally founded functional programming language. Another ML based language for the .NET platform is SML.NET [BKR04] based on Standard ML '97. L Sharp.NET [Bla07] is an implementation of the Lisp functional programming language for the .NET platform. DotLisp [Hic07] is a lisp like interpreted .NET language.
Turning from formally founded functional language compilers for .NET to non-.NET compilers for process oriented languages, we find that the BoPi language reported in [CLM05] implements a distributed computing semantic based on asynchronous π-calculus i.e. the send and receive prefixes are asynchronous. Compared to this, the π-ADL.NET implementation is based on synchronous π-calculus. A compiler implementation also exists for the occam-pi language [BW05], which embraces elements of both CSP and π-calculus.
The PiLib [CO03] domain specific language is an interesting alternative solution for providing a process coordination abstraction in that it is implemented as a software library hosted by the Scala language. This delivers the functionality to program process oriented constructs without having to implement a separate compiler and runtime – that of Scala is used. It is notably relevant to the π-

ADL.NET work that this implementation posed the researchers a set of challenges in adapting the semantic style of Scala for the purpose of supporting PiLib. However the contrast in semantics and the type system between Scala and PiLib isn't nearly as great as that between π-ADL and CIL. Also, the application domain of PiLib is primarily as a simple π-calculus interpreter designed for teaching purposes. The π-ADL.NET work, of course, has different goals. During the implementation of π-ADL.NET we realized the limitations imposed on the design, had a high level language such as C# been used to host π-ADL.NET instead of the assembly level CIL. The most notable limitation is the legal character choices in variable naming, which is the same for π-ADL and C#, and supports no easy resolution of conflicts between the naming of internal and user defined variables.
Amongst modeling tools for ADLs, the Honeywell® MetaH has a workspace environment for which the source module components [Met08] are of relevance to this work. They allow the generation of some aspects of the executable from architectural specifications written in MetaH.
A distantly comparable work is also available in the form of the analysis and constraint checking tools for AcmeStudio [Acm08], the eclipse based modeling environment for the Acme ADL.
Thus we find .NET compilers for functional languages, compilers for π-calculus based languages and parser tools for ADLs. While the work reported in this chapter may be related to all of these three categories of research, it forms a distinct domain of work in itself by virtue of being the only π-calculus based ADL compiler for the .NET platform.

Chapter 5: Testing Results for π-ADL.NET

This chapter presents my experiences in the development, testing and usage of the π-ADL.NET compiler. The objectives of this study are two-fold:

- To help developers define the scope of the implementation problem for similar projects, in the domain of compilers or other language processing tools for π-ADL or other π-calculus based languages.
- To help software testers articulate comprehensive testing cases for the problem domain.

Sections 5.1 to 5.3 describe important experiences acquired in this regard during the implementation, testing and usage of these features. Section 5.4 presents a conclusion.

5.1 Parallel Processing Constructs

By specification, a compose or a choose block can contain an arbitrary number of sub-blocks. Additionally, each of a compose, choose, or replicate block can have other compose, choose, or replicate blocks inside them, as well as if and while blocks. And in the case of pseudo-applications of abstractions within a compose, choose, or replicate block, the correct function of all of these blocks needs to be verified from within the abstraction. As can be seen, there are many permutations for nested blocks that needed to be tested in order to assure their correct function to a desired degree. Table 5.1 represents the test cases examined for parallel processing constructs.

Syntactic Element	Test Cases
compose	• Less than two sub-blocks • Two or more sub-blocks • Compose, choose, replicate, if and while blocks nested inside one or more sub-blocks • Compose, choose, replicate, if and while blocks nested inside one or more abstraction pseudo-applied from within this block
choose	• Less than two sub-blocks • Two or more sub-blocks • Compose, choose, replicate, if and while blocks nested inside one or more sub-blocks • Compose, choose, replicate, if and while blocks nested inside one or more abstraction pseudo-applied from within this block
replicate	• Compose, choose, replicate, if and while blocks nested inside replicate

Table 5.1: Test Cases for Parallel Processing Constructs

Since there is no limit on nestation levels, more elaborate test cases such as a replicate block inside a compose block embedded within a choose block were executed.

5.2 Connections

While the actual connection operations are easy to describe, their correct function is influenced by a number of factors:

- The atomicity of send and receive operations. This means that no concurrent thread should be able to access or modify a variable while it is being sent or received.
- Connection mobility. Connections can be sent and received through other connections, and can be passed as arguments to abstractions. This means that both the sending and receiving ends of the

connection share the same class definition. Earlier compiler implementations created a new MSIL connection class against each π-ADL connection declaration, and then instantiated it for use only once. This meant, for example, that a connection being sent from one abstraction to another abstraction could not be identified on the receiving side since it was not aware of its class definition. To overcome this limitation, a generic connection class was created. The type flexibility this solution offers is sufficient to overcome the previous limitation since connections can differ only in they type they can transfer, and have otherwise identical semantics for all operations.

- The connections must provision for having multiple send operations queued up. Receive operations must execute without the loss of any data sent in such a case.
- Connections also need to cater to the synchronization needs of choose statements in-case the first statement of a choose sub-block is a receive statement. What this means is that if all sub-blocks in a choose block each have a receive statement as their first line of code, then the competition amongst them will be decided in the favour of the first block to have its receive operation completed.

Based on these usage scenarios, a number of test cases had to be developed with the objective of validating the usage scenarios presented in Table 5.2.

Syntactic Element	Usage Scenarios
send	• Send values • Send variables • Send from within a compose sub-block with a corresponding receive in a parallel sub-block • Send without a corresponding receive in parallel (to test indefinite blocking) • Send via the special out connection, which never blocks • Send with a parallel receive within an abstraction being pseudo-applied • Send through a connection through multiple parallel threads without a single corresponding recieve (to test queueing) • Test compiler error detection by sending a variable or value different from connection type • Send over a renamed connection
receive	• Receive values • Receive variables • Receive from within a compose sub-block with a corresponding send in a parallel sub-block • Receive as a first statement in one or more or all sub-blocks within a choose block • Recieve without a corresponding send in parallel (to test indefinite blocking) • Recieve via the special in connection, which includes runtime type validation as well • Receive with a parallel send within an abstraction being pseudo-applied • Recieve through a connection through multiple parallel threads without a single corresponding send (to test queueing) • Test compiler error detection by receiving a variable different from connection type • Receive over a renamed connection

Table 5.2: Test Case goals for Connections

Note that each one of these usage scenarios may have many different manifestations under the influence of other testing scenarios. For example, when testing the correct sending over a renamed connection, we need to test for both values and variables, with or without a corresponding receive, or with the receive operation performed within an abstraction being pseudo-applied, and so on. We note that without finding logical exclusions in the number of permutations of the above mentioned usage scenarios, the total possible number of resulting test cases could be roughly n-factorial or more, for usage scenarios related to each of send and receive. Therefore while we have tested hundreds of different ways of validating the send and receive implementation in π-ADL.NET, we do not yet claim to have covered all possible contexts of executions for these elements. And yet the next section will explain how still a lot more test cases remain to be covered for testing connection semantics.

5.3 Constructed Data Types

For all data types in π-ADL, be they primitive or constructed, a certain number of basic operations are guranteed. These operations are:

- Declaration
- Representation as a value and as a variable
- Assignment, as left- or right-value
- Be sent or received via a connection
- Passed as an argument to an abstraction via a pseudo-application

In addition π-ADL.NET ensures that all data types can be sent and received via the special *in* and *out* connections.

Type	Declaration Syntax	Examples
tuple	"tuple [" valueType ("," valueType)* "]"	t1 : tuple[String, Float, tuple[Boolean, Integer]];

view	"view [" identifier ":" valueType ("," identifier ":" valueType)* "]"	v1 : view[a : String, b : Boolean];
		v2 : view[a1: Boolean, b1 : sequence[view[a2: Boolean,
		b2 : String]];

union	"union [" valueType ("," valueType)* "]"	u1 : union[String, view[b : Boolean, f : Float], Integer];
any	"any"	a : any;

sequence	"sequence [" valueType "]"	s1 : sequence[Integer];
		s2 : sequence[view[a: Boolean, b : String]];

Table 5.3: Constructed type declarations

Type	Declaration Syntax	Examples
tuple	"tuple" "(" (value ",")* value ")"	tuple("Hello VALORIA", 17.08, tuple(true, 42))
view	"view" "(" (identifier ":" value ",")* identifier ":" value ")"	view(a : false, b : "False val") view(a1 : false, b1 : sequence(view(a2 : true, b2 : "string 1.1"), view(a2 : false, b2 : "string 1.2")))
union	value	<constant of one of the union member types>
any	value	<constant value of any type>

sequence	"sequence (" value (, value)* ")"	sequence(1,1,2,3)
		sequence(view(a : false, b : "False val"), view(a : true, b : "True val"))

Table 5.4: Constructed type constant value format

Although this list of operations is common amongst all elements, for the purposes of testing our implementation three significant differences need to be considered:

- Each data type has a unique format for declaring variables of its type. Table 5.3 lists the syntax with examples for each one of the constructed data types. Similarly, as illustrated in Table 5.4, constant values corresponding to each constructed data type also have type specific syntactic structures.

- In π-ADL all assignments and parameter passing to an abstraction is done by value. This happens easily for primitives, but for constructed types, the implementation of this feature is very specific for each type i.e. for each of the types tuple, view and sequence etc., a separate deep copy function was written. These functions are inherently recursive since each one of these constructed types can contain other constructed types. Furthermore, the semantics of the union entails that in the role of an l-value it accepts one of its member types instead of a union of identical signature; and as an r-value its usage is valid only within a type-select block (explained below) where its currently assigned type is well established, and it acts exactly like a variable of its currently assigned type. Similarly the by value transfer in send and receive operations, and passing as argument in an abstraction pseudo-application requires individually catered implementations for each of the constructed types.
- The correct implementation for the in and out connections specific to the π-ADL.NET implementation also varies widely for each constructed data type. For example, the input of a view or tuple from the console requires the system to prompt the user for the value of each member seperately. The input of a sequence entails that the user is queried about the number of elements to be input, and the based on that the user is allowed to enter values a specified number of times. The input of a union and any comprises the specification of the type by the user, followed by the entry of the value of that type. All aspects of the input are validated for all constructed types, and in case of error the user can try again. The output is formatted in the same manner as specified in Table 5.4 for each of the constructed data types, except for union and any types, the formats for which are shown in Table 5.5.

Type	Output format	Examples
union	[type1, type2, ..., currentType*: value, ..., typen]	[String, Integer*:45, Boolean]
any	[currentType : value]	[String : "Hello world"]

Table 5.5: Union and any output format

We now review testing scenarios specific to individual data types.

5.3.1 Tuple

Besides the common testing scenarios, a tuple also supports projection via the syntax `"project" tuple as c1, c2, ...`. The types of c_1, c_2, ... must correspond to the constitutent types of the tuple in the correct order. The projection is a by-value operation. For example: `project t as a, b;`. Therefore we note that for a tuple, the testing of the projection operation was also required. The following additional usage scenarios apply:

- Projection into a correctly ordered list list of variables with types corresponding to the tuple signature
- Projection into an incompatible list of variables

5.3.2 View

The view also supports projection, as well as another access operation named shorthand projection, which allows access to its individual elements. This operation takes the syntactic form `view "::" cn` as an l- or r-value. For example for a view named v with a boolean member b: `v::b = false;`. Therefore the additional usage scenarios for a view are:

- Projection into a correctly ordered list of variables with types corresponding to the tuple signature

- Projection into an incompatible list of variables
- Short-hand projection with existing members of the view
- Short-hand projection with non existant members of the view
- Perform all applicable operations to short hand projection that are common to all the variables, as well as those specific to the type of the variable being projected

Note that in the case of a short-hand projection, recursive cases in which a member of a view that is itself a member of another view and so on also needed to be tested. Note also how the last testing item compounds the number of test cases to be considered.

5.3.3 Union

While a union supports the standard operations common amongst constructed data types, it differs in the scope of some of those operations. Declaration, assignment to a union l-value, recieving a union, and I/O via the in and out connections can take place anywhere. However a union itself cannot be assigned to another union in its declared form. Instead, the current value of the union can be assigned as an r-value within the context of a type select-case block. The select-case block takes the following form:

```
selectBlock := "select " union "{" caseBlock "}"
caseBlock := (" case " valueType " do " statementBlock)+
```

For example, Listing 5.1 tests a union for possible current value types.

```
select u {
  case String do
    via out send u;
    via out send "\n";
    u = "New string value";
  case Integer do
    via out send u;
    via out send "\n";
    u = 42;
}
```

Listing 5.1: A Type Select Case block

Note that besides enabling the r-value assignment operation, the type select-case block morphs the union type into a variable of its currently selected type. For example, in listing 5.1 the union u is considered as a string or integer in all operations involving it. For the first case, sending u via the out connection will result in an ordinary string being printed, instead of the usual formatted output mentioned in table 5.5.
Another operator that union supports is the type equality operator with the syntax #=. For example, for two union type variables u1 and u2, the following code will display "Equal" if both are holding the same current data type:

```
if (u1 #= u2) do
    via out send "Equal\n";
```

Note that both u1 and u2 need not have an identical signiture for this operator to work. Based on the above description, the following additional usage scenarios needed to be validated:

- Correct behaviour of the union for all applicable operations in their proper contexts (within and outside select-case blocks).
- Correct working of the select-case blocks.

- Behaviour in case when a type case matches the current union type.
- Behaviour in case when a type case does not match the current union type.
- Morphing of union into its current type within a select-case context.
- Use of the type equality operator covering type matches and mismatches

5.3.4 Any

As already mentioned, an any type can be considered as an open union with no restriction of valid types. Consequently, it demonstrates the same behaviour as a union within a type select-case block, and can also be an operand to the type equality operator. The following usage scenarios needed to be validated:

- Correct behaviour of the any for all applicable operations in their proper contexts (within and outside select-case blocks).
- Correct working of the select-case blocks.
- Behaviour in case when a type case matches the current any type.
- Behaviour in case when a type case does not match the current any type.
- Morphing of any into its current type within a select-case context.
- Use of the type equality operator covering type matches and mismatches.
- Use of the type equality operator with the opposing operand as a union, or as an any.

5.3.5 Sequence

Besides the general usage cases, a sequence mainly adds the indexing operations of being able to read or write one of its members using a zero-based index. One important implementation feature is that if the index being accessed exceeds the current count of values within the sequence, the sequence resizes automatically up to that index, and inserts default values of the type for the intermediate indices. On the other hand when attempting to read an out-of-bounds value, the runtime will generate an error. Consequently the following usage scenarios needed to be considered:

- Update a member of a sequence indexed within its current bounds.
- Update a member of a sequence indexed outside its current bounds.
- Read a member of a sequence indexed within its current bounds.
- Attempt to read a member of sequence index outside its current bounds.

```
ci : sequence[     view
                   [a1: Boolean, b1 : sequence[view[a2: Boolean, b2 : String]      ]
        ] ];
cii : sequence[    view
                   [a1: Boolean, b1 : sequence[view[a2: Boolean, b2 : String]      ]
        ] ];
//...
cii(0)::b1(1)::b2 = ci(0)::b1(1)::b2;   //various combinations of reading
                                        //and writing tested
```

Listing 5.2: Example using nested views and sequences

As with other types, recursive cases also needed to be tested, as shown in Listing 5.2. The impact of recursive definitions on testing needs here, as elsewhere, was very significant and was compounded when combined with short-hand projections.

5.4 Conclusion

As seen in section 5.1 and 5.2 in general, and section 5.3 in particular, mutually perpendicular language features tend to quickly compound the number of cases needed to correctly test a language compiler, or some other language processing tool. This has important implications for architectural considerations in designing a compiler. For example, the particular nature of connections in π-ADL led me to eventually define a generic class to closely represent the language concept while staying independent of the specific type of data passed through it. Similarly, the compile-time and runtime bifurcation for implementing constructed data-types evolved as the exact implications of their usage became clearer.
Turning to the testing methodology employed in this work, this work does not pursue any formal approach towards software verification. While these techniques provide a proof derived certainty lacking in conventional case-based testing, the formidable amount of effort required to do perform formal verification often makes it impractical [SC06]. The path of software testing itself is bereft with a permanent lack of certainty about usage scenarios that weren’t covered [Dij72], yet it often presents the only viable option for assuring software quality.

Chapter 6: .NET Extensions to π-ADL

This chapter proposes language extensions to π-ADL in order for it to interface with .NET software libraries. As discussed previously, the motivation for the π-ADL.NET project is to provide a large experimental space for testing and evaluating the process oriented, formally founded π-ADL in the context of a widely used software development and execution platform. At the same time there is a need to opportunistically enhance the π-ADL language syntax in order to leverage the functionality available in the reusable .NET software libraries. The .NET platform is fundamentally object-oriented, and the need to represent .NET semantics while remaining within the confines of the process-oriented paradigm of π-ADL poses a modeling problem. Section 6.1 provides syntactic details of the proposed language extensions. Section 6.2 presents a case study that puts the extensions to π-ADL to work, highlighting their utility in widening the modeling horizon of π-ADL. Section 6.3 presents related work with a comparison to the .NET API access features in other .NET languages based on formal methods. Section 6.4 discusses the impact of the extensions on the architecture oriented nature of π-ADL. Section 6.5 concludes this chapter.

6.1 The Extensions

We now present the proposed syntax extensions to π-ADL to provide support for accessing the .NET API. All details are presented from the π-ADL perspective, using elements defined in π-ADL to model the extensions. An understanding of the .NET language framework is assumed.
The focus of this current set of extensions is to allow the use of existing .NET libraries in π-ADL.NET, and not develop reusable components in π-ADL.NET for access through other .NET languages. The latter could form the focus of later work on π-ADL syntax extensions.

6.1.1 Declaring and Instantiating Classes

Let NET be the .NET governor behaviour, in terms of the π-ADL formalism. We declare that NET has connections type_in and type_out, and they both process values of type any. NET supports a typing system expressed by the .NET namespace value attached to each type it receives or sends e.g. "System.String". The π-ADL type system is therefore extended to be able to evaluate the .NET namespace notation and hence recognize the .NET types. All .NET objects are treated as abstractions in π-ADL.
The instantiation of a .NET class would be as follows:

```
x:System::Text::StringBuilder; //declaration
via NET::type_in send "System::Text::StringBuilder";
via NET::type_out receive x;
```

Here the type_in connection receives a string value indicating the type of object it will process. The NET behaviour then internally creates the object and sends it out via the type_out connection. A shorthand notation forming the language syntax for this would be an overload of the '=' sign connoting instantiation, with the restriction of using parenthesis after the class name:

```
x:System::Text::StringBuilder; //declaration
//instantiation:
x = System::Text::StringBuilder();
```

x would now contain a reference to an abstraction of type System::Text::StringBuilder.

6.1.2 Namespaces

In order to recognize the .NET namespaces and incorporate them for usage in π-ADL programs, we specify the namespaces being used in a π-ADL program using the *use* directive. It is used at the program level, outside behaviour or abstraction definitions. Therefore:

```
//... declaration(s)
x = System::Text::StringBuilder();
```

can be rewritten as:

```
use System::Text;
//...behaviour header
//... declaration(s)
x = StringBuilder();
```

Note that the namespace System is implicitly used and need not be declared using the use keyword.

6.1.3 Constructors

The statement

```
x = System::Text::StringBuilder();
```

would result in NET internally calling the default constructor of StringBuilder. In case a constructor with arguments needs to be called, we can employee NET::type_in as follows:

```
//invoking the StringBuilder constructor
// with an Integer argument defining
//initial length
//... declaration(s)
via NET::type_in send
       "System::Text::StringBuilder";
via NET:: type_in send 42;
via NET:: type_out receive x;
```

Generally, the arguments to the constructor will be sent via type_in to the NET behaviour, right after sending the string that contains the type of object that needs to be created. The shorthand notation to this syntax would be:

```
x = System::Text::StringBuilder(42);
```

Multiple arguments will be sent as a comma separated list e.g.

```
y = MyObject("arg1", true, 0.5);
```

6.1.4 Methods, Fields and Properties

In order to incorporate the use of methods in π-ADL, we develop an internal model of a .NET class that is recognizable in the π-ADL syntactic domain, and covers interaction with both static and non-static fields and methods. Each .NET class is represented by a corresponding behaviour, and serves to provide access to static fields and methods. Conforming with π-ADL syntax this behaviour contains publicly accessible variables (for representing static fields).
Static methods are treated as connections. Each connection receives an argument of type any. It also makes a return value available via an input prefix (for methods that have non-void return values). In order to model non-static fields and methods, a similar approach is used with the difference that abstractions and not behaviours represent an object.

Listing 6.1 shows a .NET class and Listing 6.2 shows what it would look like in extended π-ADL. Notice that a .NET class is fully expressed by a behaviour and an abstraction, to model the static and non-static aspects of the class respectively. The following example shows how fields and methods for .NET objects are accessed in π-ADL:

```
x : PiADL::Vector; y : Float;
x = PiADL::Vector(3,4);
compose {
      via x::Resultant send Void;
and
      via x::Resultant receive y;
}
```

```
public class PiADL.Vector {
  private double _x, _y;
  public double X {
    get { return _x; }
    set { _x = value; }
  }
  public double Y {
    get { return _y; }
    set { _y = value; }
  }
  public Vector(int x, int y) {
    _x = x;  _y = y;
  }

  public static string Name()
  {
    return "Vector";
  }
```

```
  public static string Space()
  {
    return "PiADL";
  }

  public double Resultant()
  {
    return Math.Sqrt(_x*_y);
  }

  public double Angle()
  {
    return Math.ATan(_y/_x);
  }
} //end class
```

Listing 6.1: .NET class (using C# code)

```
Vector names behaviour
{
  Name : Connection[any];
  Space : Connection[any];

  compose
  {
    replicate {
      via Name receive;
      via Name send "Vector";
    }
  and
    replicate {
      via Space receive;
      via Space send "PiADL";
    }
  }
} //end behaviour

value VectorInstance is abstraction {
  X : Float; Y : Float;
```

```
Resultant : Connection[any];
Angle : Connection[any];
dVal : Float;

compose
{
    replicate {
      via Resultant receive;
      dVal = X * Y;
      dVal = Math.Sqrt(dVal);
      via Resultant send dVal;
    }
  and
    replicate {
      via Angle receive;
      dVal = Y / X;
     dVal = Math.ATan(dVal);
      via Angle send dVal;
    }
} //end abstraction
```

Listing 6.2: π-ADL model for the .NET class

The shorthand equivalent of the above code would be

```
x : PiADL::Vector; y : Float;

x = PiADL::Vector(3,4);
//shorthand for the method call compose
y = x::Resultant();
```

Fields and properties are treated as shorthand projections e.g.

```
x::Length = 42;
```

There is also a case of indexed properties, which are treated from the programmer's perspective as an array. Both arrays and indexed properties are accessed using square brackets enclosing the index value, such as:

```
a : System::Collections::ArrayList;
o : System::Object;
//initialize and populate a
o = a::Item[4];
```

6.1.5 Events

Abstractions handle events generated by .NET objects. The handles keyword will allow the assignment of an abstraction to handle a certain event type. The proposed syntax is:

```
use  System::Windows::Forms;
behaviour {
      t : TextBox;
      t = TextBox();
      t::Click = a;
}
value a is abstraction (e : EventArgs)
      handles TextBox::Click {
//...
}
```

Events are modeled as connections. The assignment of the abstraction `a` to the event `t::Click` can be taken as shorthand for the following two statements:

```
via t::Click receive e;
via a send e;
```

Where `e` in this case is of type `EventArgs`. This syntactic expansion is considered to be located in a compose block along with other event delegations. Note that the parameter to the abstraction must be the same type as that generated by the event. In case of multiple parameters being generated by the event, the handling abstraction would receive the parameters as named members of a view. Also as is the case with a pseudo-application, the possibility of unifying a connection from the event generator component to the event handling abstraction exists. It therefore necessitates suitable syntax for connection renaming as depicted in the following example:

```
t::Click = a where {x renames y};
```

6.1.6 Enumerations

A .NET enumeration is modeled as a π-ADL quote type, which was designed to serve the same purpose.

6.1.7 Exceptions

Since exceptions are full-fledged classes in .NET, they can be treated using the same behaviour-abstraction model described in 6.1.4 above. Examining the exception properties and variables, and calling its methods is the same as described in 6.1.4. The .NET implementation of exceptions entails the intervention of the platform runtime in case of an exception, which allows the termination of the try block code at the point where the exception occurred, and transfer of control to the catch block. Consequently we model exception handling in π-ADL terms as follows:

$$\frac{e : Exception \quad P_1 : \mathrm{Pr}ocess \quad P_2 : \mathrm{Pr}ocess \quad P_E : \mathrm{Pr}ocess \quad P = (P_1; P_2) \quad P_1 \wedge e}{try\ P\ catch\ P_E \rightarrow P_1; P_E}$$

The above model can be described as follows: There are two processes P_1 and P_2, which when composed sequentially result in the process P that represents the code in a try block. *e* is an exception object, and P_E is the process representing the collection of associated catch blocks. If P_1 is responsible for generating the exception e, then the P-P_E try-catch block becomes a sequential composition of P_1 and P_E.
The exception handling syntax based on this model is the try-catch-finally approach seen in C#, as demonstrated below:

```
s : String;
i : Integer;
e1 : FormatExeption;
e2 : OverflowException;
try {
      via out send "Enter an integer: ";
      via in receive s;
      i = Convert::ToInt64(s);
      i = i * i;
      via out send i;
}
catch (e1, e2)
{ via out send "Invalid input value."; }
finally {
  via out send
    "\n\nEnd of try-catch-finally demo.\n";
}
```

Multiple catch blocks can be associated with a try block. Compose or choose blocks are not allowed inside the scope of try, catch or finally blocks. The reason for this is that the underlying .NET platform does not have a mechanism for consolidated handling of exceptions arising from multiple threads of execution, and only supports sequential, synchronous exceptions.

6.1.8 Null Values

.NET objects initialized as Null values can be recognized and compared using the null keyword. All .NET objects are assumed null at the time of declaration. The null value is not a valid value for any of the π-ADL base or constructed data types.

6.1.9 Delegates

.NET delegates are represented using the connection renaming syntax in π-ADL. For example we apply this proposed syntax for delegates to the class definition in Listing 6.2:

```
m : Connection[any];
m renames Vector::Name;
```

Since methods receive arguments and return values that are not necessarily of the same type, modeling delegates in terms of an `any` connection represents the only viable choice within π-ADL.

6.1.10 Casting to and from System::Object

System::Object is the canonical root class from which all .NET classes are derived. Consequently, .NET collection types often cast into System::Object the elements they are collecting. In the π-ADL.NET compiler, each of the basic types corresponds to one of the primitive types of the .NET platform. The .NET platform provides the ability to box primitive types into representative objects e.g. the System::Int64 structure exists for 64-bit integers and so on. In π-ADL.NET the constructed types are implemented as .NET classes and hence are directly cast-able to and from System::Object. Therefore the following proposed syntax for casting π-ADL types to and from objects is easy to implement:

```
c : System::Collections::ArrayList;
v : view[a : String, b : Boolean];
o : System::Object;
i : Integer;

c =  System::Collections::ArrayList();
i = 5;
v = view(a : "Cast test", b : true);

c::Add(i); //implicit cast to System::Object

o = (System::Object)v; //explicit cast
c::Add(o);
v = (view)c[1]; //explicit cast
```

6.1.11 Upcasts and Downcasts

Following the syntactic convention in 6.1.10, .NET objects can be cast to any one of their inherited types and back. As such casts are dynamically checked in .NET, any runtime errors resulting from incorrect casts will have to be handled using the exception handling mechanism discussed in 6.1.7. The following example illustrates the syntax:

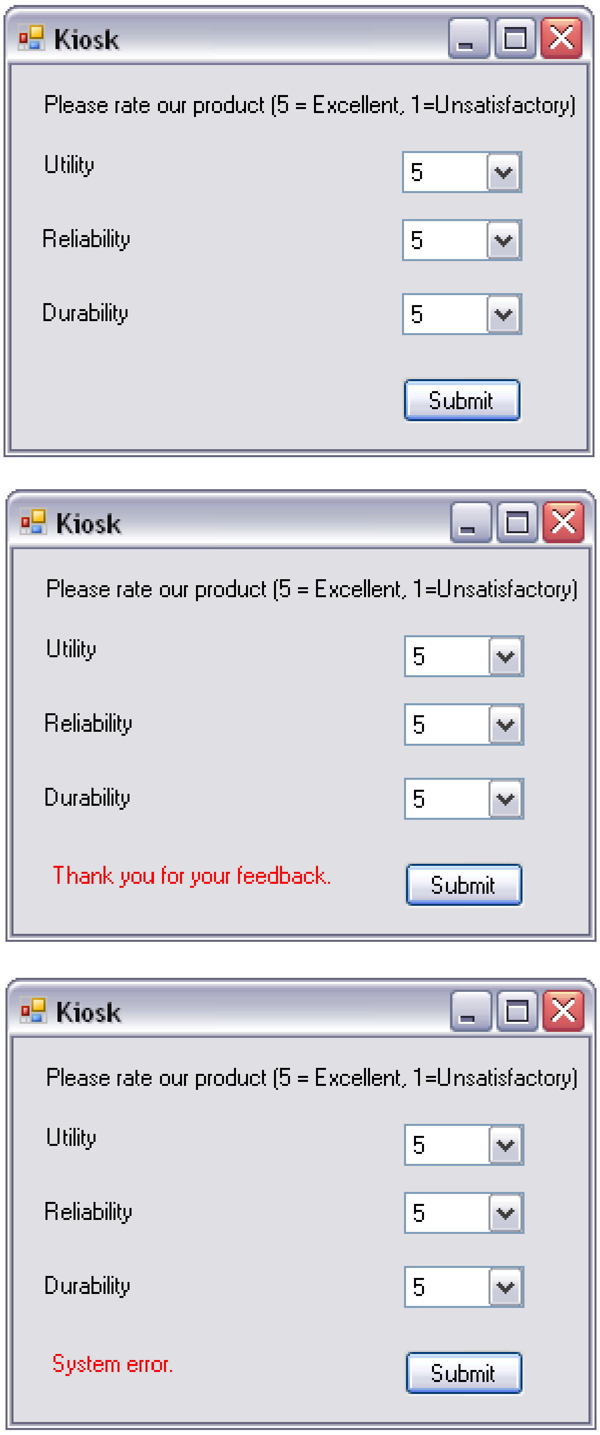
```
use System::Windows::Forms;
behaviour {
	b : Button; c : Control; o : Object;
	b = Button();
	c = (Control)b; //upcast
	o = (Object)c; //upcast
	c = (Control)o; //downcast
	b = (Button)c; //downcast
}
```

6.1.12 Generics

Since our current focus in .NET extensions for π-ADL.NET is to provide the ability to use existing class libraries only, and not to be able to create new ones, we need not go into a detailed syntax mapping for .NET generics. It is sufficient to be able to declare and instantiate a .NET generic class as follows:

```
c : GenericClass<Integer>; //declaration
c = GenericClass<Integer>(); //instantiation
```

For generic class with multiple generic parameters, the parameters can be comma-separated. From the π-ADL perspective, the class GenericClass<T> represents a family of behaviour-abstraction pairs, each one of which processes a certain data type. Note that this model is compatible with constrained generic classes as well [BKR04]. π-ADL should also be able to use its own basic and constructed types as type parameters when instantiating a generic.


Kiosk

Please rate our product (5 = Excellent, 1=Unsatisfactory)

Utility	5
Reliability	5
Durability	5

Submit

Kiosk

Please rate our product (5 = Excellent, 1=Unsatisfactory)

Utility	5
Reliability	5
Durability	5

Thank you for your feedback.

Submit

Kiosk

Please rate our product (5 = Excellent, 1=Unsatisfactory)

Utility	5
Reliability	5
Durability	5

System error.

Submit

Figure 6.1: A Simple three-tier Kiosk Application

6.1.13 Inner classes

Just as a regular .NET class is fully defined by its name and namespace, an inner class can be defined by its name, container class and namespace. From the π-ADL.NET perspective a public inner class is treated in the same manner as an ordinary class. For example if a class Container contains the class Inner, and is declared in the namespace MyNameSpace then the following π-ADL.NET code will be valid:

```
i : MyNameSpace::Container::Inner;
```

```
i =  MyNameSpace::Container::Inner();
```

The π-ADL.NET code then can access the members of Inner like for any other class or object.

6.2 Case Study

The case study we present here is a hypothetical Kiosk software system that presents existing customers with a product feedback platform. The user can assign values between 1 and 5 to three product parameters of Utility, Reliability and Durability. Upon pressing the Submit button, this information is stored along with the current date, and a unique record number in a Microsoft Access database. The user is informed of the success of the operation or if a system error is produced. Figure 6.1 shows the three possible states of the user interface.

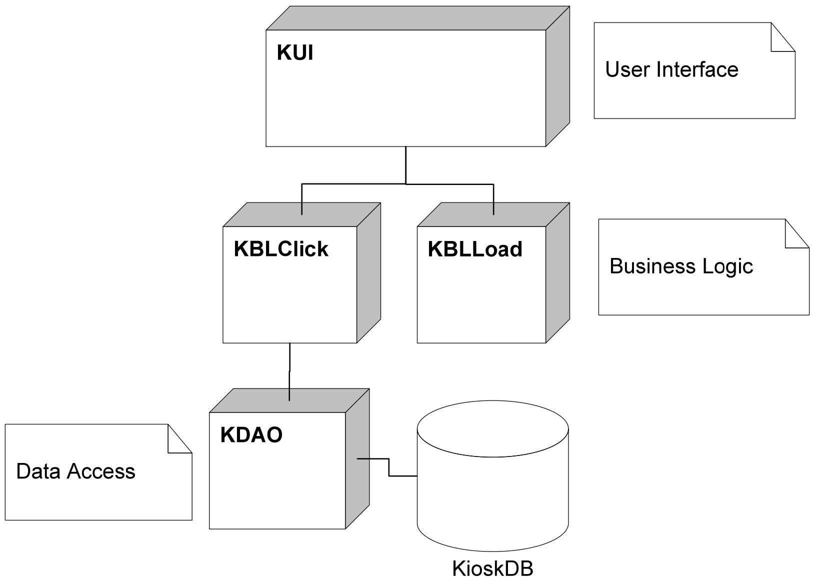


Figure 6.2: The architecture of the Kiosk application

Although the Kiosk application is very simple, it provides the context for demonstrating the use of many of the .NET extension to π-ADL that we propose in this paper. Figure 6.2 gives a diagrammatic representation of the software architecture of the Kiosk application. The π-ADL implementation of this architecture entails that KUI would be declared as a behaviour while KBLLoad, KBLClick and KDAO are abstractions. Logically we can refer to KBLLoad and KBLClick as composing the business layer of the Kiosk architecture. Table 6.1 summarizes the functionality associated with each one of the components. Note that we are not concerned with the implementation of the database component KioskDB since it is designed using MS Access. This database consists of a single table and its schema is shown in Table 6.2.

We now present the implementation of each of the functions mentioned in Table 5.1.

6.2.1 Declare and initialize UI components

Using syntax descriptions in sections 6.1.1 to 6.1.4, we declare UI components and initialize their values. The code below gives some representative excerpts of the UI design code:

```
use System::Windows::Forms;
use System::Drawing; //For using the Point
                     //and Size classes
```

```
Use System::Data::OleDb; //for database I/O
KUI names behaviour {
bView : view[utility : Integer, reliability
       : Integer, durability : Integer,
       inputTime : String];
bConn : Connection[union[Boolean, String,
       view[ utility : Integer, reliability
       : Integer, durability : Integer,
       inputTime : String]]];
bSignal : Boolean;
bMessage : String;
frm : Form;
//...
lblNote : Label;
cboUtility : ComboBox;
//Declare other variables...
lblNote = Label();
lblNote::Location = Point(13, 13);
//Initialize other properties...
cboUtility = ComboBox();
cboUtility::Items::Add("1");
cboUtility::Size = Size(60, 21);
//Initialize other properties...
frm::Controls::Add(lblNote);
frm::Controls::Add(cboUtility);
//Add other controls...
Application::Run(frm); //standard .NET
  //method for running a forms application
//...
```

KUI	• Declare and initialize user interface components (labels, drop down boxes, and a button), and set their initial properties • Associate handlers with the page load, and button click events • Display messages to the user originating from KBL
KBL	• Perform functionality related to the page load and button click events • Transfer user input to the KDAO component • Message the user interface regarding the success or failure of database operation
KDAO	• Perform database insert operation • Notify KBL of success or failure of insert

Table 6.1: Functional roles of different components of the Kiosk application

Column Name	Type
ID	Auto incrementing integer
Utility	Byte sized integer
Reliability	Byte sized integer
Durability	Byte sized integer
InputTime	Date

Table 6.2: The Feedback table in KioskDB

.NET user interface design is very straightforward and we were able to implement the user interface of the Kiosk application with repeated application of just a few extensions. We see the application of declaration, initialization, use of constructors, and method and property access. We also demonstrate the use of a static method using the Run method of the Application class. The variables bConn and bView are declared for message passing to the BL, and will be discussed later.

6.2.2 Associate event handlers

This is a direct application of the event handling syntax, implemented in KUI:

```
//our form object is declared as frm, and
//the Submit button as btnSubmit
frm::Load = KBLLoad;
btnSubmit::Click = KBLClick;
```

6.2.3 Handling events in business logic

The two abstractions KBLLoad and KBLClick are declared as follows:

```
value KBLLoad is abstraction (arg : view
       [sender : object, e : EventArgs])
       handles Form::Load {
       //perform any load time
       //operations here
}
value KBLClick is abstraction (arg : view
       [sender : object, e : EventArgs]) handles Button::Click {
   //implementation details in IV.D and F
}
```

As discussed before, the event handling qualification of an abstraction is embedded in its header declaration. Notice the use of a view argument to the abstraction in order to encapsulate multiple parameters sent by the event generator.

6.2.4 Channeling data into DAO

This function consists of two operations: receive data from KUI, and send it to DAO. Our approach to receiving data from KUI is to develop a protocol of exchange depicted in Figure 6.3.

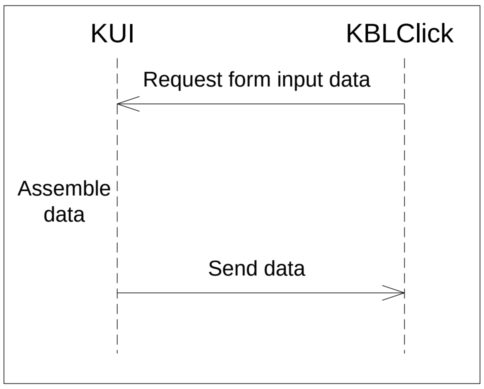


Figure 6.3: Data exchange between KUI and KBLClick

The following code represents the part of the logic encoded in KBLClick responsible for transferring user input to the DAO from KUI:

```
bView : view[utility : Integer, reliability
       : Integer, durability : Integer,
       inputTime : String];
bConn : Connection[union[Boolean, String,
       view[ utility : Integer, reliability
       : Integer, durability : Integer,
       inputTime : String]]];
dConn : Connection[Boolean];
dValue : Boolean;
via bConn send true;
via bConn receive bView;
bView::inputTime = DateTime::Now::
                    ToShortDateString();
via KDAO send bView where {dConn renames dConn};
//...
```

The connection bConn demonstrates the flexibility of communication we can achieve by using a Union type. We are able to recycle the same connection for transferring a Boolean signal and then a more elaborate data structure. The following code shows the KUI side of logic for completing the protocol depicted in Figure 6.3, and also demonstrates the utility of a union connection. This code sequentially appears right after the code shown in 6.2.1:

```
Via bConn receive signal;
bView::utility = Convert::ToInteger
             (cboUtility::SelectedItem);
bView::reliability = Convert::ToInteger
             (cboReliability::SelectedItem);
bView::durability = Convert::ToInteger
             (cboDurability::SelectedItem);
Via bConn send bView;
//...
```

6.2.5 Perform Database Insert

The following code performs the legwork for database I/O:

```
value KDAO is abstraction (b : view[utility:
  Integer, reliability : Integer, durability
  : Integer, inputTime : String]) {
  dConn : Connection[Boolean];
  cn : OleDbConnection;
  cmd : OleDbCommand;
  query : String;
  e : Exception;
  query = String::Format(
    "INSERT INTO Feedback ( Utility, Reliability, Durability, InputTime ) \
    VALUES ({0}, {1}, {2}, '{3}')",
    b::Utility, b::Reliability,
    b::Durability, b::InputTime);

  cn = OleDbConnection(
    "Provider=Microsoft.Jet.OLEDB.4.0;Data Source='kioskdb.mdb'");
  cmd = OleDbCommand( query, cn);
  try {
      cn::Open();
      cmd::ExecuteNonQuery();
  }
  catch (e) {
      via dConn send false;

  }
  via dConn send true;
} //end abstraction KDAO
```

In order to communicate with KDAO, a Boolean connection dConn is used. The abstraction code is straightforward in that it directly uses API functionality to achieve a database insert operation, with the last section demonstrating the use of a try-catch block. The communication of Boolean values via dConn in both cases serves to determine the value to be displayed to the user, as discussed next.

6.2.6 Escalating messages to the user

As seen in the KDAO code, a Boolean value is sent via the dConn connection indicating the success or failure of the operation. There is a corresponding Boolean connection dConn declared in KBLClick which is unified with the dConn in KDAO, as shown in the KBLClick code listing in 6.2.4. The following KBLClick code immediately follows that listing:

```
via dConn receive dValue;
if (dValue) do
  via bConn send
      "Thank you for your feedback.";
else do
  via bConn send "System error.";
```

The following KUI code follows the KUI code listing in 6.2.4:

```
via bConn receive bMessage;
lblResult::Text = bMessage;
```

6.2.7 Analysis

Our case study allowed us to examine the practical usage of the basic .NET syntax extensions to π - ADL, as well as cover exception handling and events. More importantly, it gives us a baseline of comparison with existing .NET languages. The case study was simultaneously implemented using C#, allowing us to better appreciate the level of platform integration that our extensions provide vis-à-vis that language. Barring the limitation of not being able to compile π-ADL with .NET extensions, our proposed syntax additions seem to be well validated by this case study. Although the case study application is trivial, it is structured in conformance to an important architectural design that puts the message passing and modularizing nature of π-ADL to good use.

6.3 Related Work

The work presented here merits a comparison with a similar work performed for other formally founded languages. For example the F# language [SM07] compiles to the .NET platform, and is able to access .NET components. Both π-ADL and F# are based on formal methods, although F# is a functional language while π-ADL is a process oriented language. But in each case, the foundations of these languages differ greatly from that of .NET, which is principally object-oriented. Furthermore F# provides syntax level interoperability for all the .NET features discussed in section 6.1 except for generics, thus providing an almost equivalent level of access to the .NET API, when compared to the π-ADL .NET extensions. However F# has been designed as a .NET language right from the start, whereas π-ADL as a language is neutral to any platform technologies.
SML.NET [BKR04], based on Standard ML '97 [MTR97], supports most of the features discussed in section 5.1, but does not provide support for events or generics.
L Sharp.NET [Bla07], which is an implementation of the Lisp functional programming language for the .NET platform, provides support for namespaces, object instantiation, and access to static and non-static methods, fields and properties. Advanced features such as casting, event handling, exceptions and generics are however not supported. DotLisp [Hic07], a lisp like interpreted .NET language has limited support for .NET types and delegates, as well as exception handling. It does not allow the instantiation of objects or the use of namespaces.
In short when it comes to compilers and extensions to formally founded languages for .NET, the current body of related work restricts itself to functional languages. Plus as seen in this section, none of these languages provide support for generics, an important feature of .NET version 2.0 [Low05]. Seeing this reported work from another angle, there is no Architecture Description Language compiler for .NET besides π-ADL.NET. This work is thus a significant overture in that it provides the

possibility of examining a process-oriented, formally founded Architecture Description Language in the context of and in interaction with the .NET universe.

6.4 Conforming to Architectural Properties

The design of π-ADL ensured that architectures described using it fulfil the architectural properties of decomposability, interface conformance and communication integrity. The extensions proposed in this chapter enable the possibility of manipulating objects within behaviours and abstractions, raising the question as to whether their presence may affect the form of the architecture. Here we note that the extensions only allow the creation and manipulation of objects, and do not propose the creation of classes from which they are derived. Further, no means of object or abstraction discovery is available to the object, so it cannot form connections within the system outside the architectural form. From a component point of view, any liason of an object with others through instantiation can be regarded as strictly internal behaviour, and has no consequence for the architectural form. Thus communication integrity and interface conformance is preserved, regardless of the internal structure of an object employed in the architecture. Under our model, decomposability is implicitly present in the implementation when seen as an evolution of the architecture.

6.5 Conclusion

The purpose of the .NET extensions to π-ADL is to provide syntactic basis for the π-ADL.NET compiler to interact with .NET libraries. At the same time, these extensions form a reference for interfacing π-ADL with other software technologies. The development of these extensions is particularly important in that the architecture oriented focus of π-ADL lends naturally to the notion of employing ready-made software components to put together a software system, or perhaps roll out a software development project in which both architecture and components evolve through interaction with each other. As demonstrated through our case study, the reach of an ADL is enhanced considerably when employed in tandem with detail oriented components. The purpose here is not to make ambiguous the separation of concerns, but enable greater freedom and new possibilities in software architecture design.
One goal of the π-ADL.NET project is to allow software architectural modeling on a mainstream software development platform. The implementation of these proposed extensions will enable the development of software using multiple paradigms and languages simultaneously. This helps meet the universal goals of reliable and low-cost software development through the following advantages: effort put in defining the software architecture is employed directly in the resulting software system; the privilege of an architectural view of the system that can be analyzed through compilation and execution; and benefits of the architectural paradigm are brought to the rich technological foundations of .NET.

Chapter 7: A π-ADL.NET Case Study in Service Oriented Architectures

In this chapter a comprehensive application of π-ADL.NET in describing a software architecture is presented. Previously π-ADL has been applied in the domains of grid computing [MMO05], human computer interface applications, knowledge management, and industrial process automation [Oqu06]. This latest application in service oriented architectures is a good test of the usefulness of π-ADL.NET in that it provides the opportunity to test π-ADL in a new context. It establishes that the compiler performs well when describing architecture types that have before not been a subject of π-ADL based modelling. This chapter first presents the background concepts (section 7.1), followed by problem definition of the case study (section 7.2). Section 7.3 presents the π-ADL implementation. Section 7.4 concludes the chapter.

7.1 Background Concepts

This case study has been developed in collaboration with Marcos López-Sanz who at the time of this writing is associated with the Kybele Research Group at Rey Juan Carlos University. The software architecture modelled in this study is a Short Message Service (SMS) distribution system developed to conform to MIDAS [CMV03], a methodological framework based on OMG's Model Driven Architecture (MDA) [OMG01], and developed by the Kybele Research Group. The MDA is a modelling notation that allows interoperability between different industrial software standards, with a degree of platform independence, and attempts to unify every step of development and integration from business modelling, through architecture and application modelling, to development, deployment, maintenance, and evolution. However, due to OMG's pathological persuit of platform independence, the MDA approach suffers from an inability to define the executable aspects of software systems. That is taking into account the separation in CIM, PIM and PSM levels stated by the MDA proposal, and the fact that defining executable behaviour is not explicitly supported at the PIM level under MDA. The choice of modelling the proposed SMS distribution system using π-ADL was made to fill that gap.
The problem therefore, is to model the SMS system, which is a Service Oriented Architecture, using π-ADL while remaining within the framework of MIDAS.

7.1.1 MIDAS

MIDAS proposes to divide a software system model across 3 orthogonal dimensions: *levels*, *aspects* and *phases*. The levels of the software model are content, hypertext and presentation. The content level covers the data layer of the system, which is usually encoded in a DBMS. The hypertext level is concerned with the navigation structure and the logical composition of user interfaces. And the presentation level is concerned with the graphical layout and user interaction mechanisms. The aspects are structure and behaviour. The phases are conceptual modelling, logical modelling, physical modelling and implementation. The first three phases correspond to Computation Independent Modelling (CIM), Platform Independent Modelling (PIM), and Platform Specific Modelling (PSM).
Such a division cuts through both the software development life cycle as well as role separation in design and development. However, rather than going into the details of the merits of this approach, we need to appreciate the constraints it imposes on our π-ADL based description. Our π-ADL based architecture description can be thought of fully encompassing the PIM level, and then extending that phase to cover the executable domain, conventionally modelled at the PSM level. Further, for the purpose of our architecture description, we use a meta-model defined within MIDAS for service oriented architectures. This is described in section 7.1.2.

7.1.2 Building Service-Based Architectural Models

This subsection explains briefly the elements contained in the meta-model used to define service-based architectural models at PIM level in MIDAS. The service meta-model containing the foundations of that model (Figure 7.1) can be broadly divided into four parts: service aggregators, service description and properties, service types and enumeration types.

7.1.2.1 Service Aggregators

A Service-Oriented Architecture is built upon independent entities which provide and manage services. Because SOA is widely used as a way to integrate enterprise applications, these providers serve to represent the business organizations involved in those processes into the architecture model. These providers act as service containers in charge of presenting the services contained to the world.
In the architecture model shown in Figure 7.1 these entities are identified as UML 2.0 packages. They are not considered as subsystems since they do not add or modify neither the inner services nor the dependencies or relations between them.
Service providers can be classified into two main groups: inner service providers (*innerProvider* in the metamodel), which are internal to the designed system (usually the part of the software solution whose inner elements are being designed in detail) and outer service providers (*outerProvider* in the metamodel), which are identified as external entities containing services which collaborate with the system to perform a specific task but which are not under its direct control or whose internal structure is not known or valuable for the development of the current architecture model.

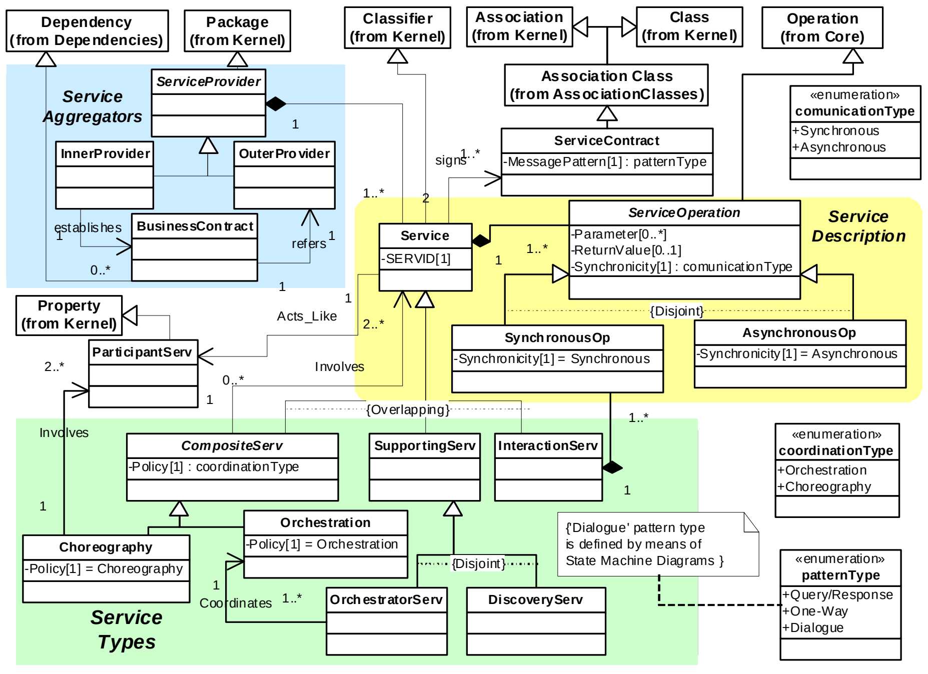


Figure 7.1: PIM-level service architectural metamodel in UML. Courtesy Marcos López-Sanz

The relationship between two service providers is modelled as a dependency understood as a 'business contract' (*bussinessContract* in the metamodel) that allows communication between the services belonging to the providers.

7.1.2.2 Service Description and Properties

Services are computational entities in charge of a resource and offer functionality associated with resource in the form of operations [FDH04]. Apart from these operations, services have another property, named *SERVID*, which allows identification of a service univocally within an SOA environment and the PIM-model. It should be possible to search, locate and access a service and therefore it must expose information for that purpose. Another reason to include an identifier inside the definition of a PIM model is that our model represents instances of services (and not classes as in usual object-oriented approaches in which class diagrams are used).

Service operations are considered as atomic functions that collaborate to build a joint description of the service. Moreover, they represent the only way of interacting with a service as they outline its interface. Their attributes include their name, parameters, returned value and operation type. It is very important to note the distinction between operation types depending on its synchronicity, as it is closely related to the type of service they take part in. In asynchronous operations (*AsynchronousOp*) the requester of the operation does not wait for the answer or return value (if any). On the other hand, in synchronous operations (*SynchronousOp*) the requester will wait for the answer or return value (that always exists). As will be mentioned later, a service with at least one 'synchronous' operation will be modelled as an interactive service (*InteractionServ*).

7.1.2.3 Service Relationships

Services relate, communicate and interact with each other through contracts (*ServiceContract* in the metamodel). The operations that take part in the contract established between services can be understood as the roles played by the contractors. In the architectural description of the service models, 'contracts' are defined as connectors, specifying point-to-point relationships between the services that subscribe to those contracts. The main property of a contract is the message exchange pattern describing the message flow between the contracted services.

The pattern for message exchange can be of three possible types: **one-way**, in which no response is expected when an operation request is made. **Query/Response**, in which there is an explicit answer to the operation requested (note that it can be sent synchronously or asynchronously). And finally **dialogue**, in which the concrete protocol can be complex and, therefore, it must be represented by means of a state machine. This explicit behavioural description must be respected by both requesting and providing services. The only constraint to the last pattern is that all the operations involved in it must be 'synchronous' in order to be able to maintain a true consistency in the communication.

7.1.2.4 Taxonomy of Services

The main role of services inside SOA is to embody functionality offered by the system. This functionality is derived from the requirements provided by stakeholders and the information gathered in the upper CIM models. The concept of services here (at PIM level) is aligned with that of the OASIS [OAS06] reference model for services:

A service is a mechanism to enable access to a set of one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.

For a better definition of the architectural view of the system, they can be classified depending on different criteria: the kind of interaction (interaction services or pure services), their atomicity (simple or composite services) or the role they play within the architectural model (basic services or supporting services).

According to the kind of interaction: Services are divided into synchronous and asynchronous types. Services always provide at least one operation. In our architectural model, services capable of performing synchronous operations will be clearly identified (and stereotyped) as interaction services (*InteractionServ* in the metamodel). Services not offering any dialogue operation need not be particularly marked as they are considered 'pure' or 'traditional' services. The interest in separating these two types of services relies on the necessity of identifying which services can maintain or initiate a conversation with any outer entity interested in the functionality provided.

According to the composition of the service: Simple services represent a behaviour of the system that does not entail the invocation of any other service defined within the system. Their functionality is satisfied entirely by means of the operations they define. On the contrary, a composite service (under the abstract class defined as *CompositeService* in the metamodel) is defined by having at least one operation in which due to its complexity or nature, it is necessary to access the functionalities offered by other services. This compound nature forces a coordination to exist among the different services taking part in the composite in order to achieve a given goal. Service coordination can be achieved with two different approaches: by means of choreographies or by using orchestration. The latter refers to the existence of a 'special' service (*OrchestratorServ* in the metamodel) in charge of carrying out a workflow defined as a complex operation involving the invocation of external service operations. The former, on the other hand, represents a coordination environment in which each service remain autonomous and in which there is no master-slave relationship. In the metamodel these two composition options are reflected by means of a property named *policy* with possible values of 'choreography' and 'orchestration'.

According to the role played in the architecture: Basic services offer operations related to the functionalities defined in the business process of the system and modelled in the architecture. However, there are some kinds of services with no direct relation to the modelled system functionality but instead provide or assist other operations necessary for the rest of the services to operate correctly. We generally refer to these services as 'supporting services' (*SupportingServ* in the metamodel) and among them we specifically identify two: orchestration services (*OchestratorServ*), that will be services in charge of performing a coordination, as detailed above; and discovery services (*DiscoveryServ*), understood as architectural entities that act as dynamicity enactors by providing the four common operations that set up a dynamic environment (link, unlink, create and destroy) [MKS89].

7.2 The Case Study

The selected case study emulates the functionality of a SMPP [SMP07] gateway system by means of services. SMPP stands for Short Message Peer-to-Peer Protocol and is a telecommunications industry protocol for exchanging SMS messages between SMS peer entities such as Short Message Service Centres. Through a service interface (Web or RPC-like) a user is able to send SMS text messages to multiple addressees. Although the system is made up by many other elements, here (see Figure 2) we focus only on the following building blocks and functionalities:

- *Reception Subsystem*: its main purpose is to receive SMS send requests directly from the user. It contains a single service (*ReceptionService*) offering several useful operations for the user such as '*sendSMS'*, which allows the user to submit an SMS send request to a set of previously stored recipients.
- *Storage Subsystem*: This subsystem stores information related to clients and SMS messages. It acts as a repository and source of information for the other two subsystems. It comprises two services:
  - *SecureDataService*: This service performs operations requiring a secure connection or the encryption of data. The supported operations include '*authenticate'* (used to verify the credentials of the client intending to send SMS messages through the gateway system) and '*updateCredit'* (used to update the financial information associated with the client depending on the action performed over the gateway).
  - *SMSManagerService*: This service is in charge of managing all the information related to SMS messages (such as status, SMS text) and creating listings and reports associated with a specific client.
- *SMS Processing Subsystem*: This subsystem is in charge of retrieving, processing and sending the SMS texts and related information to the specialized SMS server. It is made up of two different services:

  - *SMSSenderService*: This service retrieves SMS texts from the Storage Subsystem and sends them to one of the available Short Message Service Centres (SMSC).
  - *DirectoryService*: The main task performed by this service is to return the service identifier of the SMSCenterService which has to be used in order to send an SMS to a specific recipient.
- *SMSC (Short Message Service Centre):* This entity represents an SMS server capable of sending the same SMS text to a predetermined number of recipients. Its functionality is enacted by one service which receives the SMS message and the list of recipients (*SMSCenterService*).

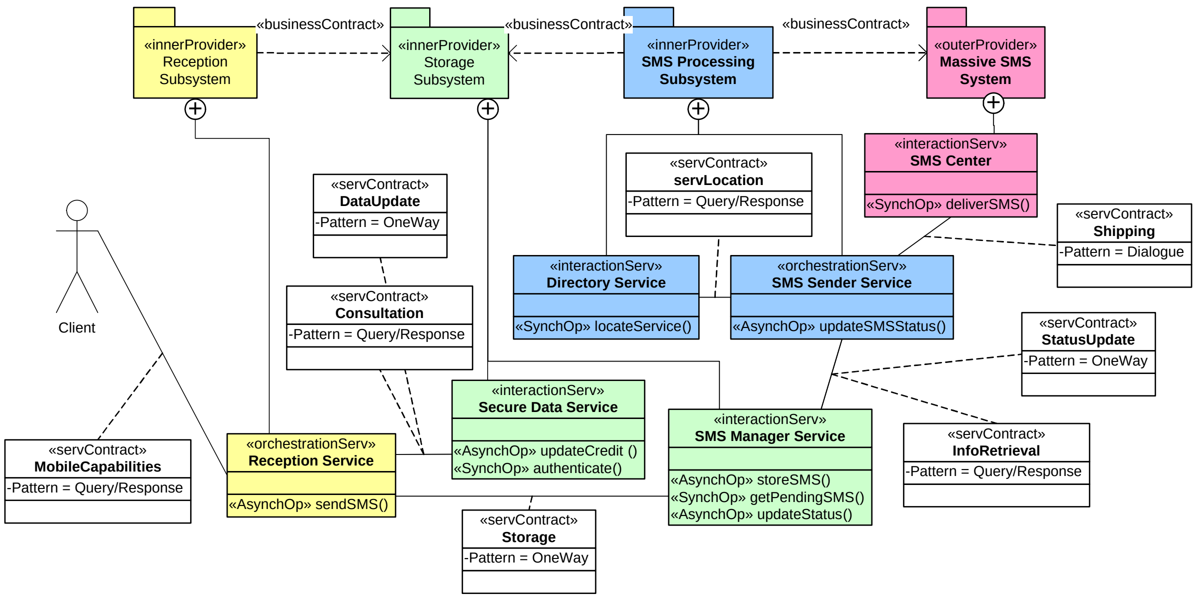

«innerProvider» Reception Subsystem
«businessContract»
«innerProvider» Storage Subsystem
«businessContract»
«innerProvider» SMS Processing Subsystem
«businessContract»
«outerProvider» Massive SMS System
«interactionServ» SMS Center
«SynchOp» deliverSMS()
«servContract» servLocation
-Pattern = Query/Response
«servContract» DataUpdate
-Pattern = OneWay
«servContract» Shipping
-Pattern = Dialogue
Client
«servContract» Consultation
-Pattern = Query/Response
«interactionServ» Directory Service
«SynchOp» locateService()
«orchestrationServ» SMS Sender Service
«AsynchOp» updateSMSStatus()
«servContract» StatusUpdate
-Pattern = OneWay
«interactionServ» Secure Data Service
«AsynchOp» updateCredit ()
«SynchOp» authenticate()
«interactionServ» SMS Manager Service
«AsynchOp» storeSMS()
«SynchOp» getPendingSMS()
«AsynchOp» updateStatus()
«servContract» InfoRetrieval
-Pattern = Query/Response
«servContract» MobileCapabilities
-Pattern = Query/Response
«orchestrationServ» Reception Service
«AsynchOp» sendSMS()
«servContract» Storage
-Pattern = OneWay

Figure 7.2: UML model of the case study

7.3 The π-ADL Implementation of the SMPP Gateway

We examine the π-ADL code for the SMPP Gateway architecture model by looking at two of the subsystems: *Reception Subsystem* and *SMS Processing Subsystem*. The entire code for these two subsystems is listed in Appendix III. The *Storage Subsystem* and the *Massive SMS System* are excluded from the analysis since they represent worker services that do not demonstrate any techniques used in this architectural model, besides some of those used in the other two subsystems. We begin by looking at the Reception subsystem, which includes the Reception Service and is the sole interaction point for the client.

7.3.1 The Reception Subsystem

Figure 7.3 shows a visual depiction of the π-ADL model of the reception subsystem. The double-headed arrows such as between `Actor` and `MobileCapabilities` indicate pseudo-applications. The reception sub-system is initialized by the behaviour `Actor`. `outConn`, `inConnectionC`, `resultConn` etc. are connections. Although π-ADL connections are bi-directional, the double-slashed side of the line connecting two connections indicates the configured direction of data flow. The cloud shows a dynamic pseudo-application, a π-ADL.NET specific feature that allows a pseudo-application over a string containing the name of an abstraction.
Listing 7.1 shows the code of the behaviour `Actor`, demonstrating only the *Send SMS* operation. This behaviour is primarily responsible for interaction with the user, and delegates operation requests to the abstraction `ReceptionService` (via `MobileCapabilities`) after prompting the user for input

using `GetUserData`. Note the unification of `paramConn` with an unrestricted connection from `GetUserData` in order receive feedback from the abstraction.
The abstraction `GetUserData`, partially listed in Listing 7.2, simply prompts the user to input SMS information, and send this information to Actor. The compose block at the end of the code is a recurrent technique throughout this case study, and allows the main abstraction to terminate and transfer control back to the calling abstraction or behaviour, even before the sub-blocks in the compose block have terminated execution. This is necessary because otherwise in a sequential communication scenario a deadlock would occur. As seen in `Actor`, the statement `via paramConn receive params` is waiting to receive from `via dataConn send data` in `GetUserData`. But in order for this input prefix to complete, the control must transfer back to `Actor` following the `GetUserData` pseudo application. Without the execution of this input prefix, the output prefix in `GetUserData` will be suspended, hence creating a deadlock.

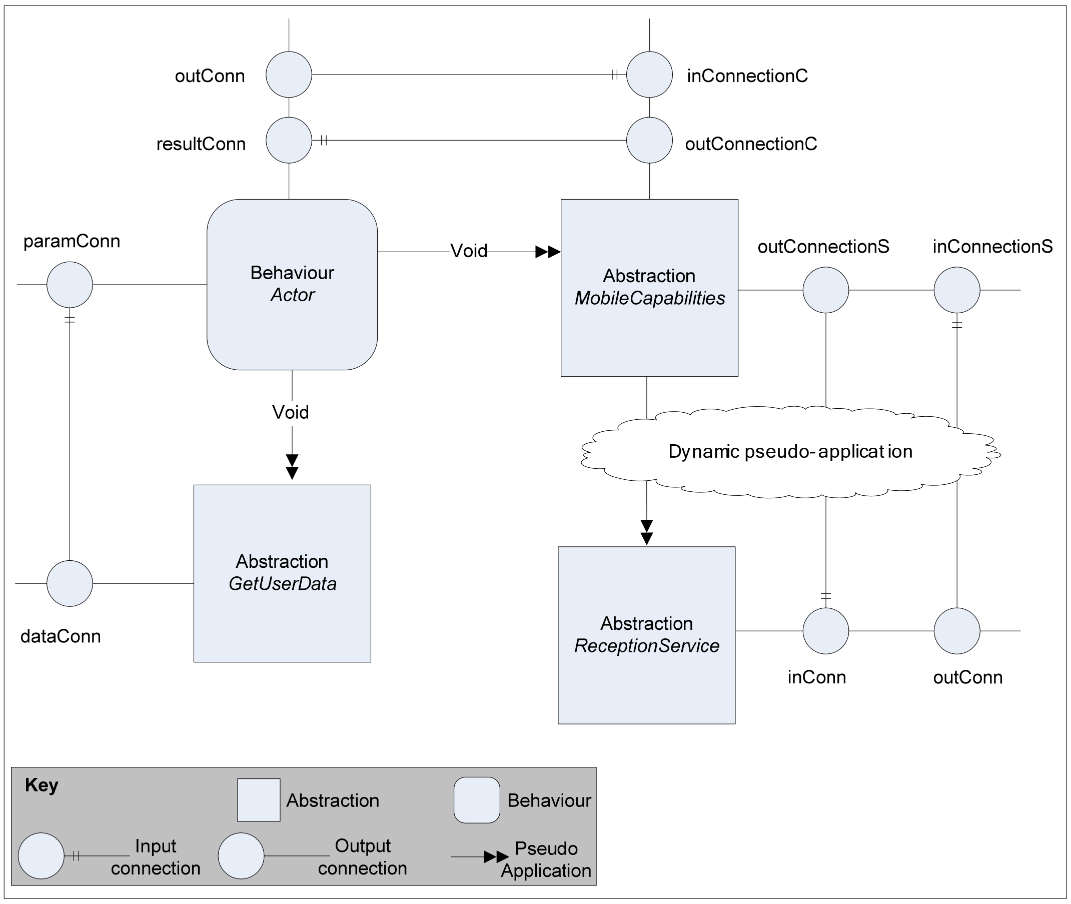


Figure 7.3: The π-ADL model of the reception sub-system

```
Actor names behaviour
{

      params : view [phoneNumber : String, simID : String, recipientList : String,
            SMSText : String];
      paramConn : Connection [ view [phoneNumber : String, simID : String,
            recipientList : String, SMSText : String] ];
      ReceptionServParams : view[operation : String, data : any];

      outConn : Connection [view[operation : String, data : any]];
      resultConn : Connection [view[operation : String, data : any]];

      result : view[operation : String, data : any];

       via out send "\n------- ACTOR starts ----";

      via GetUserData send Void where {paramConn renames dataConn};
      via paramConn receive params;

      ReceptionServParams::operation = "sendSMS";
      ReceptionServParams::data = params;


      compose
      {
            via out send "\n ----- Actor: SEND ReceptionServParams --";
            via outConn send ReceptionServParams;
      and
            via out send "\n ----- Actor: SEND MobileCapabilities --";
            via MobileCapabilities send Void where {outConn renames
                  inConnectionC, resultConn renames outConnectionC};
      and
            via out send "\n ----- Actor: waiting to RECEIVE result --";
            via resultConn receive result;

            via out send "\n ======= RESULT: ";
            via out send result::data;
      }
}
```

Listing 7.1: The Behaviour *Action*

```
value GetUserData is abstraction () {
        //data and dataConn declarations, same as params and paramConn in Action

        statusVal : Boolean;
        statusVal = false;

        while (!statusVal) do {
             //prompt user to input phone number, Sim ID, list of recipients
             //and the SMS message
             statusVal = true;  //true if phone number and Sim ID are input
                                //correctly
        }
        Compose {
             via dataConn send data;
        and
             done;
        }
}
```

Listing 7.2: The Abstraction *GetUserData*

The compose block alleviates this problem. Syntactically, a compose block must have at least two sub-blocks, thus necessitating the second sub-block that does nothing (`done` is the π-ADL equivalent of return in C#).

The pseudo-application of `MobileCapabilities` is performed within a compose block. This is an alternative and isomorphic approach to the method of establishing concurrency used from within `GetUserData`. Instead of performing a composition within the invoked abstraction, it is established within the calling unit of execution. This gives a minor syntactic advantage of avoiding a `done` statement. Two of the connections of `Actor` are unified with `MobileCapabilities`. One is used to send data to the abstraction, while the other is used for receiving data. Note that alternatively the data could have been sent as an argument to the abstraction. The use of an extra channel is necessitated because this abstraction is designed to take input from two different sources, and the type structure of the input is different in both cases. We accomplish that within `MobileCapabilities` by providing different connections for each caller, and unifying the appropriate connections at the time of pseudo application. If an abstraction caters for only one type of input, then there is no advantage using this approach over passing data using an argument.

The abstraction `MobileCapabilities` represents the service contract by the same name in the UML model in figure 7.2 between the *Client* and the *Reception Service*. A service contract defines the message exchange protocol between two services, and standardizes an interface between two services. This means that if any two services implement complementary sides of a service contract, then they can communicate under the sponsorship of that service contract. The concept can be

extended to more than two services, although in our case study we haven't had the opportunity to do so.
In a static service environment, in which contracts between services are established at design time, all the information needed by a contract to correctly fulfil its behaviour (message exchange pattern and contractors) is defined and initialized internally within the contract abstraction when the system starts. In dynamic environments however, this is normally accomplished by transferring all the information through the channel opened simultaneously when the abstraction is executed. In both cases, the contract is able to perform the behaviour needed to transfer data requests and results from one service to another from that information.
Listing 7.3 shows some highlights of the code within the abstraction `MobileCapabilities`. This abstraction concerns itself primarily with representing the protocol between two communicating services. In order to do this, a sequence of views `MessagePattern` is declared. Each view in this sequence represents a step of the protocol and contains the following fields:

- `Integer state_id`: represents the step number of the exchange protocol.
- `String via_SERVID`: stores a label identifying one of the participants in the protocol as the enactor of this step.
- `String op`: This field stores one of the two string values "send" and "receive". It indicates the type of operation that the service contract is to perform.
- `Integer numNextStates`: This indicates the number of elements in the sequence `next`.
- `sequence next`: This is a sequence of views, each of which represent a value/step number pair. The idea is to evaluate the value received from the operation (if it is a receive operation), and depending upon this value, branch the control of execution to the specified step number.

```
value MobileCapabilities is abstraction ()
{
      //declarations start...

      OneServConn :view[SERVID : String,
            inServConn : Connection[view[operation : String, data : any]],
            outServConn : Connection[view[operation : String, data : any]]];
      MessagePattern : sequence [view [state_id: Integer,
            via_SERVID : String, op : String, numNextStates: Integer,
            next : sequence[view[criteria: String, newState:Integer]]]];
      ServConnGroup : sequence [view[SERVID : String,
            inServConn : Connection[view[operation : String, data : any]],
            outServConn : Connection[view[operation : String, data : any]]]];

      //...declarations end

      //initializations start...

      OneServConn::SERVID = "ReceptionService";

      //...
      // Description of the Message Exchange Protocol
      // Message Exchange Pattern: Query/Response

      MessagePattern(0)::state_id = 0;
      MessagePattern(0)::via_SERVID = "C";


      //...

      MessagePattern(3)::op = "send";
      MessagePattern(3)::numNextStates = 1;
      MessagePattern(3)::next(0)::criteria = "sendSMS";
      MessagePattern(3)::next(0)::newState = 0;

      input::MEP = MessagePattern;
      input::ServConnGroup = ServConnGroup;
      input::numStates = 4;
      MessagePattern = input::MEP;
      maxcountState = input::numStates;
      inConnectionS = input::ServConnGroup(0)::inServConn;
      outConnectionS = input::ServConnGroup(0)::outServConn;
      S = input::ServConnGroup(0)::SERVID;

      currentState = 0;
      while (currentState > -1) do
      {
        countState = 0;
        while (countState < maxcountState) do
        {
          state = MessagePattern(countState);
          countState = countState + 1;
          if (currentState == state::state_id) do
          {
            via out send "\n\n ..... Current State: ";
            via out send currentState;
            via out send "\n ..... DATA: ";
            via out send inData;

            if (state::op == "receive") do // receive
            {
              if (state::via_SERVID == "S") do
              {
                    //receive from server
              }
              else do
              {
                    //receive from client
              }
            }
            else do
            {
              if (state::via_SERVID == "S") do{
              via out send "\n ~~~~~ waiting SEND SERVER ~~~~~";
              compose
              {
                via outConnectionS send inData;
              and
                 via dynamic(S) send Void where { outConnectionS renames inConn,
                   inConnectionS renames outConn};
              }
            }else do{
               //send to client
            }
          }

          countNewState = 0;
          maxCountNewState = state::numNextStates;
          while (countNewState < maxCountNewState) do
          {
            nextState = state::next(countNewState);
            countNewState = countNewState + 1;

            if (nextState::criteria == inData::operation) do
            {
              via out send "\n >>>>> New state ==> ";
              via out send nextState::newState;
              currentState = nextState::newState;
            }
        }
      }
    }
  }
}
```

Listing 7.3: Partial code of abstraction `MobileCapabilities`

The information structure of `MessagePattern` allows the architect to design a turing-complete state machine representing a protocol between services. After `MessagePattern` has been properly initialized, `MobileCapabilities` proceeds to execute its instructions and assists in completing the contract.

When executing the specification of a service architecture with π-ADL, every service or contract is executed as an independent thread of execution. However in order to coordinate execution the different abstractions must be linked. Because of the dynamic nature of the service architectures, contract abstractions should be reusable as the actual services they connect can vary during the lifecycle of the system. In order to achieve this flexibility, contracts (or more appropriately abstractions performing the contract role) must be able to dynamically instantiate the channel that they have to use to send or receive the data transferred in each moment. To deal with this issue π-ADL.NET defines the `dynamic(<connection_name>)` operator. This operator represents one of the main advantages for dynamic architecture specification since π-ADL allows the transfer of connections through connections. The line of code `via dynamic(S) send Void ...` represents a dynamic pseudo-application that allows the programmer to specify an abstraction name in a string and then perform a pseudo-application upon it.

7.3.2 The SMS Processing Subsystem

In order to send the SMS messages stored in the database, it is necessary to know which specific service should be used. This necessitates a specialized service attending to requests from services asking for other services to perform tasks with specific requirements. This represents a dynamic environment since it is necessary to create a communication channel that did not exist at design time but is discovered when the system is in execution (i.e. when the `SMSManagerService` must send the SMS texts to a concrete `SMSCenterService`).
In a service-oriented environment this dynamicity may occur in several scenarios e.g. when creating a new contract between services or when introducing a new service or service type into the system. In those cases it is essential to have a special service in charge of performing the usual operations that occur in dynamic environments, i.e. link, unlink, create and destroy contracts (inclusion of previously unknown services is a topic left for ongoing research). This service will be the *DirectoryService* that appears in Figure 7.2. Figure 7.4 shows the Pi-ADL architecture design for the SMS processing subsystem.

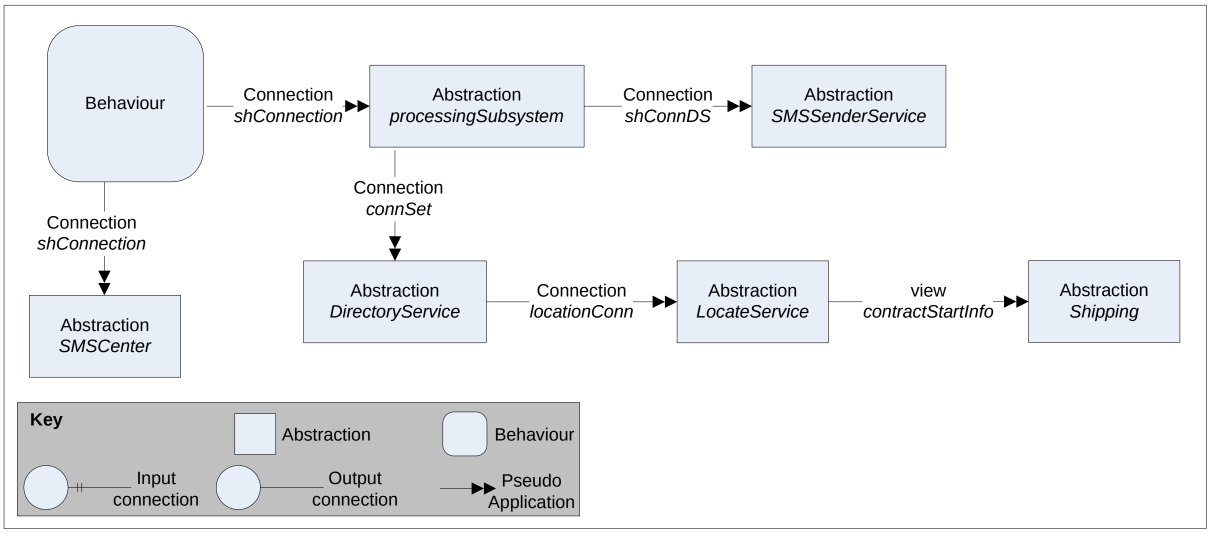


Figure 7.4: The π-ADL model of the SMS processing subsystem

The `DirectoryService` already knows the `SERVID` of any `SMSCenterService` requested. The tasks it performs when queried for a specific service include the creation of the contract needed to communicate with the `SMSCenterService` as well as the connection that will be used by both the `SMSSenderService` and the contract. The information provided by this service for the `SMSSenderService` comprises a connection to the contract allowing communication with the `SMSCenterService`. The new contract created will receive as initial information the `SERVID` of the `SMSCenterService` that it will connect with and the connection channel that should be used to communicate with it. Here, we have only focused on the information required and received in the process by each of the participants involved in the dynamism. The specific query made to the `DirectoryService` can be of three different types: query by specific `SERVID`, semantic query in which the functionality to be fulfilled is sent and query for a specific contract type to be used in the communication with another service. These three approaches would be combined to improve the functionality provided by the `DirectoryService`.

Coordination amongst services can be achieved by defining choreographies or orchestrations. Choreographies can be formalized with π-ADL by means of shared connections. Orchestrations, in turn, depend mostly on the code specified inside a unique abstraction belonging to a service playing the role of coordinator of the composition.

In our case study the only service taking the orchestrator role is the SMS Sender Service which coordinates the access to the storage subsystem (using the SMS Manager service), the retrieval of the information of the concrete *SMSCenterService* to be used to send the SMS texts by invoking the Directory Service and finally the *SMSCenterService* to complete the desired functionality.

7.4 Conclusion

MDA is one of the current leading trends in the definition of software development methodologies. Its basis lies in the definition of model sets divided in several abstraction levels together with model transformation rules. This separation in abstraction levels allows the reutilization of models at different stages of the development and favours the migration from one platform to another. This aspect is crucial when taking into account some technological approaches that have come up in the last years. The principles of the SOC paradigm and its inherent features for system integration and interoperability make MDA a suitable approach for the development of SOA solutions. In that sense, several research efforts have been carried out to cope with the complexity of defining methodological frameworks for and based on the SOC principles. One of those methodologies is MIDAS, in which we frame our research work.

While defining MDA-based frameworks, the architecture has been demonstrated to be the ideal source of guidance of the development process since it reflects the structure of the systems embedded in its components, the relations amongst them and their evolution during the lifecycle of the software being developed. In the case of MIDAS, we have defined UML metamodels for the PIM-level view of the architecture.

In this work, and in order to provide a formally founded executable model where none exists at the PIM-level in MDA-based development, we have proposed to give a formal definition of the system architecture by means of an ADL. Specifically we have chosen π-ADL because of its support for representing dynamic and evolvable architectures as well as the largely faithful compiler tool available for this language. Moreover, by using a formal representation of the system we can use mathematical formalisms to validate the UML models created for each of the abstraction levels defined within MIDAS. The whole architecture is described with the repetitive use of a few interaction patterns that are described in Section 7.3.

There are many research lines that arise from the work presented in this chapter. One research direction is – given the already defined UML notation and metamodel for the π-ADL language, the definition of transformation rules between the UML metamodel of the architecture at PIM-level and that of the π-ADL language. Another open research line is the definition of the PSM-level architectural model as well as the influence of the election of π-ADL as the ADL of choice when defining technologically dependent architectural models. More research lines include the refinement of the language support for specific SOA aspects such as the definition of choreographies, dynamic and evolvable environments requiring the representation of new types of components and connectors within the system architecture, etc.

Chapter Acknowledgements: The research presented in this chapter is partially granted by project GOLD (TIN2005-00010) financed by the Ministry of Science and Technology of Spain, the IASOMM project (URJC-CM-2007-CET-1555) co-financed by the Rey Juan Carlos University and the Regional Government of Madrid, and the Spanish project 'Agreement Technologies' (CONSOLIDER CSD2007-0022, INGENIO 2010).

Chapter 8: Conclusion

In this chapter I present a summary of my doctoral work and provide its future outlook. Section 8.1 presents an evaluation of the research work presented in this dissertation. Section 8.2 gives an outlook on the future work possibilities that arise from this project. Section 8.3 finishes this dissertation with concluding remarks.

8.1 Evaluation

Here I present in turn an evaluation each of the π-ADL.NET compiler, its testing and analysis, π-ADL extensions for .NET, and the SOA case study.

8.1.1 The π-ADL.NET Compiler

The π-ADL.NET compiler presented in chapter 3 is a unique research effort in that it's a first attempt to implement an ADL or a process-oriented language on the .NET platform, directly addressing the first three of the four points formulated in our research question:

- It succeeds in preserving the architectural integrity of the system at implementation level.
- By enabling the execution of the architecture description, it supports runtime analysis of the concrete architecture. Through suitable output messages, the architect can customize the level of analytical details presented during simulation.
- Like all text based descriptive notations, the code input to π-ADL.NET is open to incremental enhancement or evolution, and that facility is hosted by π-ADL.NET.

Although a lot of development effort went into the scanner and parser designs, these were not original contributions to computer science. On the other hand, the π-ADL to CIL mapping presented a new approach to representing a process oriented language in terms of a stack based second generation language. This includes the representation of processes (behaviours and abstractions), compose and choose blocks, data type compilation and their runtime functionality, as well as connections and their mobility.
Due to the π-ADL foundation in π-calculus, π-ADL.NET can also be regarded as a simulator for reducing π-calculus descriptions. This has mathematical and theoretical utility that goes beyond architecture specification, and can be seen as a side-effect of the architecture-centric vision of the project.
Despite a faithful representation of the π-ADL language, π-ADL.NET lacks to a very large degree the flexibility to support process location and distribution. All architecture models are developed in a single process space, with conceptual processes represented as threads. If the target implementation requires that processes be physically distributed, then π-ADL.NET based architectures become impertinent. As we see in 8.2.3, physical process distribution and separation can form the subject of future work. One the other hand, the process paradigm encoded in π-ADL and consequently π-ADL.NET is particularly relevant today with the emergence of multi-core processing technologies at the desktop and higher-end levels of computing [Sut05].

8.1.2 Testing π-ADL.NET

This work summarizes the experience of testing π-ADL.NET. It helps to generalize my implementation experience for other π-calculus based languages.

One shortcoming of this testing work is that for some critical implementations, such as connection modelling and parallel processing constructs, the application of rigorous formal verification techniques could assure that critical and often encountered code transformations are correctly coded. For example, for the generic class corresponding to connections, it should be proven that in all instances, the input and output prefixes are free of deadlocks, race conditions etc. Similarly, formal proofs could benefit the implementation of compose and choose constructs, ensuring at least partially correct kernel behaviour for all architectures developed using π-ADL.NET.

8.1.3 .NET Extensions to π-ADL

These extensions primarily address the side-effect opportunity created by the platform choice of the π-ADL.NET compiler. They can also be seen as partially addressing the suggestions of my analysis of π-ADL i.e. an object-oriented sub-system appendage to π-ADL. As vindicated by the associated case study, these extensions, despite being concerned only with component usage and not with component development, vastly increase the impact horizon of π-ADL. The fact that they amount to a complete suite of component usage syntax elevates π-ADL from a generic ADL to an actively integrated software architecture modelling solution for a mainstream platform.
This work addresses the fourth and last point of the stated research question: by integrating the programming model of the underlying platform with π-ADL syntax and applying them to the case-study, it is demonstrated that the ADL can successfully exploit the implementation mechanisms of the target platform to bring about a concrete architecture preserving system.
While the core π-ADL has been represented in π-ADL.NET, the .NET specific extensions have not yet been implemented. Therefore in the absence of tool support, we cannot establish the relevance of the .NET extensions to their stated objectives, as we can with respect to the core π-ADL.

8.1.4 The SOA Case Study

This case study serves multiple purposes. It acts as a significant test case for the π-ADL.NET compiler. It is also the first application of π-ADL in modelling architecture in the service oriented domain. The specialized form taken by our MIDAS based system of services provided a unique challenge in architecture modelling, and as has been noted, resulted in a set of new π-ADL specific patterns. It demonstrates the tight correspondence between architectural style and ADL syntax, and highlights how a certain ADL is capable of directing the way a system is modelled. Further, it validates the claims made in 8.1.1 related to our initial research question, through a demonstrable implementation.
Despite providing an executable prototype, this case study falls short of detailed implementation. The reason is that the physical distribution characteristics associated with services are not represented in π-ADL.NET, as discussed in 8.1.2. Also, in the absence of an implement for platform specific extensions, access to environmental resources such as data base connections cannot be modelled.

8.2 Future Work

An advantage of research based on tool development is the creation of further research possibilities in subsequent enhancements as well as applications. The work reported here is centred on the π-ADL.NET compiler, and consequently we explore the opportunities of future research as it applies to further enhancing the modelling environment for π-ADL. Figure 8.1 shows the future direction of this research vis-à-vis its current state. A significant number of research, development and enhancement pathways are open for the short, medium and long term, starting with improvements in the π-ADL.NET compiler itself.

8.2.1 Short term projects

- ***Optimization***: Although the π-ADL.NET compiler produces a reasonably small and robust executable size that suits our experimental purposes and has been proven to be scalable, a significant amount of optimization work can still be done, both for compilation, and for runtime efficiency of the executable. For short term projects, standard techniques for loop, dataflow and SSA-based optimization can be applied.
- ***Debugging***: Another possible improvement to π-ADL.NET is to provide debugging support to the CIL output that represents executable code. This entails providing a unique incremental label for every line of CIL code. The problem is complicated by the fact that there already exists a selective labelling scheme that provides support for control structures and such. Introducing comprehensive debugging support would require the replacement or integration of the existing scheme with the required labelling system.
- ***Implementing .NET Extensions***: One obvious direction in future work is to implement the .NET extensions proposed in chapter 6. Compared to core π-ADL concepts, these extensions will be much easier to implement since they map directly onto the underlying .NET object model.

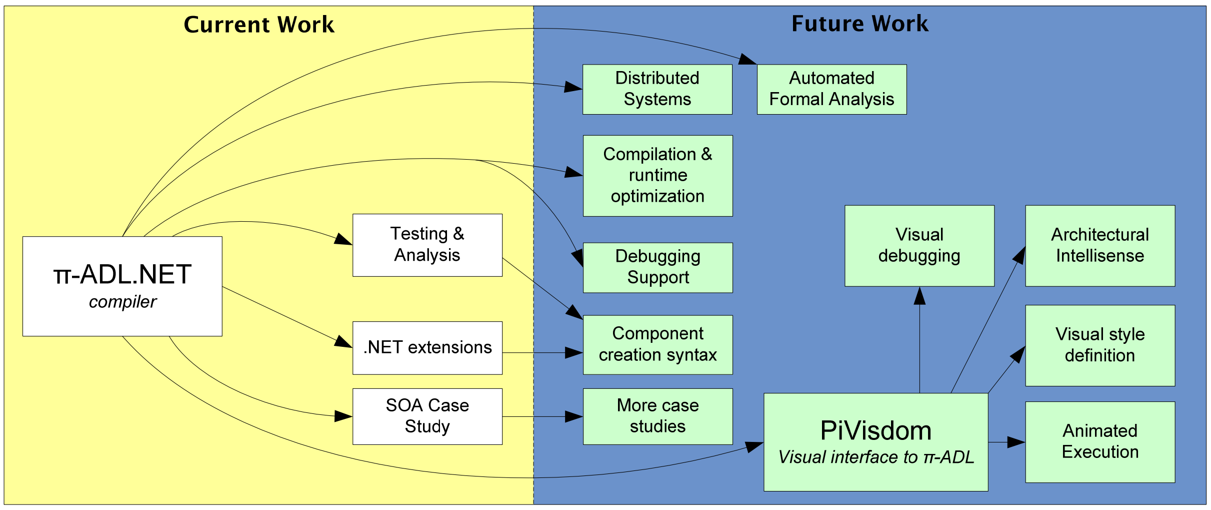


Figure 8.1: Current and future research on π-ADL.NET

8.2.2 Medium term projects

- ***A Complete OO Subsystem***: Future directions of research also exist beyond the .NET specific extensions. An object oriented sub-system within π-ADL, that will allow the language to define all the entities discussed in chapter 6 instead of simply instantiating and using them, can form an interesting subject of future research. The objective of this is to explore any possible augmentation to expressivity and ease of use, without undermining the advantages of a small syntactic base and formal foundations of the language. Such a system within the context of our .NET implementation will provide a concrete and applied example of a process and object oriented ADL hybrid. This also opens the possibilities of experimenting with architectural approaches towards component design, as well as implications for architecture driven development.
- ***π-calculus Simulator***: As discussed already, π-ADL.NET can be used to simulate the reduction of π-calculus descriptions of systems. By developing a translator of standard π-calculus to π-ADL with relatively little effort (since the underlying concepts are identical for both notations) and integrating it into π-ADL.NET, mathematicians can be provided with a convinient simulation environment for their π-calculus descriptions.

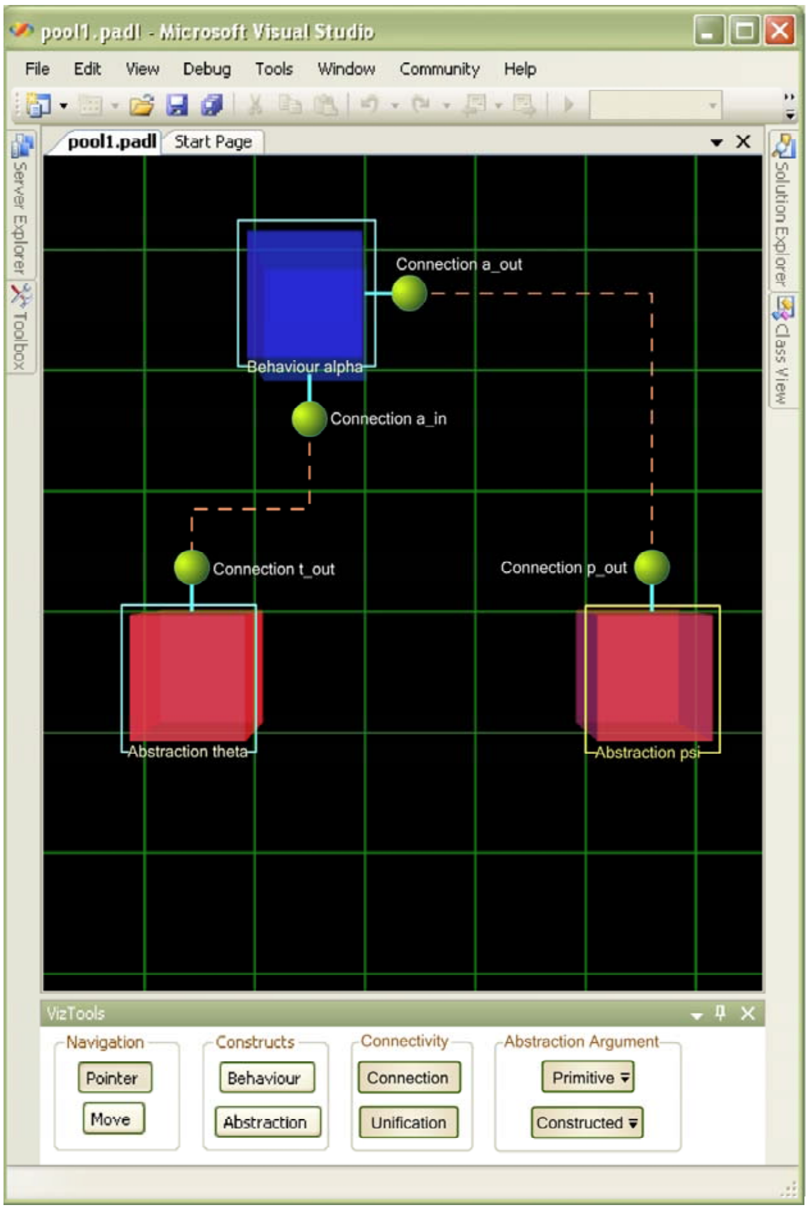


Figure 8.2: PiVisdom proof of concept work

8.2.3 Long term projects

- ***Visual Modelling***: From my current point of view, the next logical step in facilitating a development context for π-ADL is to provide a visual modelling interface for this language. Towards this end, the PiVisdom project has been initiated. This project aims to provide a 3D modelling and programming interface for π-ADL.NET, using Microsoft DirectX and .NET technologies. Figure 8.2 shows a screenshot of a current prototype developed as a plug-in to Microsoft Visual Studio .NET. The following objectives have been laid out:
  - Support the construction of hierarchical models and provide drill down functionality at the behaviour and abstraction level.
  - Develop multiple graphical notations for different architectural styles which can be used for applications in their domain, while conforming to the π-ADL base syntax [Oqu05].
  - Develop an intelli-sense system for assistance in describing software architectures, in order to design well structured, error free software architectures.

  - Develop a system for simulating the execution of the architecture in order to visualise system behaviour at the architectural level.
  - The environment will facilitate integration of component implementations developed using appropriate languages such as C#,Visual Basic, C++ etc.
  - Support a library of architectures which can be used as base for developing newer, more specialized software architectures.
  - Integrate the modeller into Microsoft Visual Studio .NET as a plug-in in order to complement and benefit from the functionality of the IDE
- ***Formal Analysis***: In parallel with the development of π-ADL, a formal analysis language called π-AAL was designed in order to support automated verification and model checking of software architectures described using π-ADL [MO06]. This language is based on the μ-calculus and combines predicate and temporal logic to facilitate analysis of both structural and behavioural properties. At the moment there is no tool implementation that applies π-AAL to π-ADL.NET. A π-AAL.NET project can be conceived to complement π-ADL.NET in order to provide automated tools for the desired formal verification support.
- ***Distributed Systems:*** The current implementation of π-ADL.NET models processes as threads of execution at the operating system level. This is an obvious constraint when implementing distributed systems, since concepts such as multiple system processes, multiple virtual machines, and multiple hosts are not represented. In order to introduce these implementation level concepts, additions to both the language syntax and the π-ADL.NET compiler will be needed:
  - A system for annotating behaviours and abstractions in π-ADL will need to be introduced in order to precisely encode process location.
  - Standard network communication protocols will need to be employed in order to transparently implement inter-process communication across multiple hosts.
  - The π-ADL process location information will need to be mapped to the universal resource locations of implementation level processes
- ***Applications and Case Studies***: The π-ADL.NET environment could be validated with further case studies, and conversely, more and more application areas could benefit from π-ADL based modelling within an executable context, resulting in a library of various architectural styles. There is current work in modelling grid-computing systems based on π-ADL using π-ADL.NET. Other service oriented systems are also being experimented with.

8.3 Concluding Notes

In 1990 Rodney Brooks wrote a paper titled **Elephants Can't Play Chess** [Bro90], lamenting the then current state of stasis in artificial intelligence research. In that paper Brooks contended that the symbol system hypothesis upon which classical AI is based is fundamentally flawed, and imposes severe limitations on the viability of research arising from it.

> **Further, we argue that the dogma of the symbol system hypothesis implicitly includes a number of largely unfounded great leaps of faith when called upon to provide a plausible path to the digital equivalent of human level intelligence. It is the chasms to be crossed by these leaps which now impede classical AI research.**

Brooks then goes on to introduce an alternative view to which he still adheres: the *nouvelle AI*. This view is based on the physical grounding hypothesis which states that to build a system that is intelligent, it is necessary to have its representations grounded in the physical world. An intuitive generalization of this hypothesis would be that in order to develop functionally successful systems, they must be developed in symbiosis with their target environment.
By "physically grounding" π-ADL into an implementation, I have attempted to achieve a similar objective within the context of software architecture research. Former work on architecture description languages implicitly avoided implementation support in ADLs in order to preserve the architectural identity of the proposed language. This work demonstrates that by extending the formal,

architecture centric paradigm embodied in π-ADL into the implementation, that identity can still be retained. Furthermore, it gives clarity and greater relevance to architectural constructs when imbued within the implementation. For concrete implementations realized using π-ADL.NET, the architectural properties of decomposability, interface conformance and communication integrity are automatically fulfilled when the architecture becomes the implementation. And it is hoped that as a scientific tool, π-ADL.NET also acts an enabler for software researchers in concretizing their hypotheses and thought experiments.

Appendix I: π-ADL.NET BNF Specification

Note: In this specification, I have used the single quote “'” to denote a comment line. Besides that, the standard BNF notation is followed.

```
' --- Program
program := [explicitBehaviourDeclaration | abstractionDeclaration]+

' --- Types
typeDeclaration := valueTypeDeclaration | namedExplicitBehaviourDeclaration
valueTypeDeclaration := typeName ":" valueType ";"
typeName := [typeName "::"] identifier
inputConnection := "in"
outputConnection := "out"
identifier := letter lettersNumbersUnderscores
letter := (A-Za-z)
lettersNumbersUnderscores := (A-Za-z0-9_)*
type := valueType | BehaviourType
valueType := baseType | constructedType | connectionType
baseType := Void | float | string | boolean | integer
Void := “”
float := [sign] unSignedFloat
unSignedFloat := unSignedInteger ["." unSignedInteger]
unSignedInteger := [0-9]+
sign := + | -
string := " (^")* "
boolean := true | false
integer := [sign] unSignedInteger
value := string | float | boolean | integer | "" | tupleValue | viewValue | unionValue | anyValue | sequenceValue
constructedType := tuple | view | union | any | sequence
tuple := "tuple" "[" valueType ("," valueType)* "]"
tupleValue := "tuple" "(" (value ",")* value ")"
view := "view" "[" identifier ":" valueType ("," identifier ":" valueType)* "]"
viewValue := "view" "(" (identifier ":" value ",")* identifier ":" value ")"
union := "union" "[" valueType ("," valueType)* "]"
unionValue := "union" "(" valueType "::" value ")"
any := "any"
anyValue := value
connectionType := Connection "[" [valueType] "]"
behaviourTypeDeclarationPrefix := identifier " names "

' --- Behaviours
behaviourDelcaration := explicitBehaviourDeclaration | implicitBehaviourDeclaration
explicitBehaviourDeclaration := namedExplicitBehaviourDeclaration | unnamedExplicitBehaviourDeclaration
namedExplicitBehaviourDeclaration := behaviourTypeDeclarationPrefix "behaviour" "{" behaviourBody "}"
[renamingClause]
unnamedExplicitBehaviourDeclaration := "behaviour" "{" behaviourBody "}" [renamingClause]
implicitBehaviourDeclaration := "{" behaviourBody "}" [renamingClause]
behaviourBody := declarations (statementBlock)* terminalBlock

' --- Statements
declarations := (restrictStatement | connectionDeclaration | valueTypeDeclaration)*
argument := [restrictStatement | connectionDeclaration | valueTypeDeclaration]
restrictStatement := "restrict" connectionDeclaration
connectionDeclaration := typeName ":" connectionType ";"
prefix := outputPrefix | inputPrefix | silentPrefix | matchPrefix | pseudoApplication
outputPrefix := "via " typeName | outputConnection " send " (typeName | value) ";"
inputPrefix := "via " typeName | inputConnection " receive " typeName ";"
silentPrefix := "unobservable;"
matchPrefix := "if (" logicaExpression ") do " (done | unobservable | prefix | assignment | "{" statementBlock "}")
whilePrefix := "while (" logicaExpression ") do " (done | unobservable | prefix | assignment | "{" statementBlock "}")
done := "done ;" / /generates opcode ret
unobservable := "unobservable ;"
```

```
renamingClause :=  "where {" [connectionRename]* lastConnectionRename "}"
connectionRename := lastConnectionRename  ","
lastConnectionRename := identifier "renames" identifier ‘identifier must be a connectionType
projection := "project" constructedType "as" typeName ("," typeName)* ";"

' --- Expression
expression := logicalExpression
logicalOp := relOp | andOp | orOp | notOp
notOp := "!" ‘precedence 1
relOp : = > | >= | < | <= | == | != ‘precedence 2
typeEqOp := #= ‘precedence 2
andOp := && ‘ precedence 3
orOp := || ‘precedence 4
logicalExpression := orFactor [ orOp logicalExpression ]
orFactor := andFactor [ andOp orFactor ]
andFactor := ([ notOp ] relFactor | notOp "(" relFactor ")") [eqOp relFactor]
notFactor := ( "(" logicalExpression")" | boolean )
relFactor := "(" logicalExpression ")" | polymorph typeEqOp polymorph | (arithmeticExpression relOp
arithmeticExpression) | (string eqOp string) | boolean

' --- Arithmetic
arithmeticOp := addOp | multiplyOp
addOp := + | -
multiplyOp := * | / | %
arithmeticExpression := addFactor [arithmeticExpressionRHS]
arithmeticExpressionRHS = addOp addFactor [arithmeticExpressionRHS]
addFactor = term [addFactorRHS]
addFactorRHS = multiplyOp term [ addFactorRHS ]
term := [addOp] (bool | string | integer | float | "(" arithmeticExpression | logicalExpression ")")

' --- Assignment
assignment := identifier "=" expression ";" 'valid for basic and constructed types

' --- Blocks
behaviourHandle := (enclosedBlock | BehaviourType)
enclosedBlock := "{" block "}"
block := (chooseBlock | composeBlock | replicateBlock | statementBlock | selectBlock)
terminalBlock := (chooseBlock | composeBlock | replicateBlock)
chooseBlock := "choose {" block [" or " block]+ "}"
composeBlock := "compose {" block [" and " block]+ "}"
replicateBlock := "replicate " behaviourHandle
statementBlock := (prefix | assignment | projection)*
selectBlock := "select " union "{" caseBlock "}"
caseBlock := (" case " valueType " do " statementBlock)+

' --- Comment
comment := "//" AnyTypeOfCharacters Newline

' --- Abstraction
abstractionDeclaration := "value " identifier " is abstraction ( " argument " ) " enclosedAbstractionBody
enclosedAbstractionBody := "{" behaviourBody "}"
pseudoApplication :=  "via " Abstraction " send " (typeName | value) [renamingClause] ";"
dynamicPseudoApplication := "via dynamic(" string ") send " (typeName | value) [renamingClause] ";"
```

Appendix II: Code Listings of View, Union and Sequence Data Types Runtime Implementations for π-ADL.NET Executables

II.1 PiConstructedType

```
/*************************************************************************************
*************************************************************************************
   Class PiConstructedType: This class is a direct parent to the View, Tuple and
 Sequence classes and provides methods common to these data types. It is also parent
              to Polymorph which is the base class of Union and Any.
*************************************************************************************
************************************************************************************/

    public class PiConstructedType
    {
        public PiConstructedType()
        {
        }

        public bool isConstructedType(string type)
        {
            if (type == "Union" || type == "Tuple" || type == "View" ||
                type == "Any" || type == "Sequence")
                return true;
            else
                return false;
        }

        public string NetToPiADLTypeName(string type)
        {
            /*string, float64, bool, int64, Tuple, View, Union, Connection,
Sequence*/
            switch (type)
            {
                case "string":
                    return "String";
                case "float64":
                    return "Float";
                case "int64":
                    return "Integer";
                case "bool":
                    return "Boolean";
                case "Connection":
                    return "Connection";
                default:
                    if (isConstructedType(type))
                        return type.ToLower();
                    break;
            }
            return string.Empty;
        } //end NetToPiADLTypeName

        public virtual bool deepCompare(PiConstructedType t)
        {
            return false;
        }
    }
```

II.2 View

```
/*********************************************************************************
**********************************************************************************
                                    Class View
**********************************************************************************
*********************************************************************************/

    public class View : PiConstructedType
    {
        public ArrayList TypeData; //always strings
        public ArrayList Values; //any base or constructed type
        public ArrayList Names; //always strings

        public View()
        {
            TypeData = new ArrayList();
            Values = new ArrayList();
            Names = new ArrayList();
        }

        public int addType(string type, string name, object val)
        {
            TypeData.Add(type);
            Values.Add(val);
            Names.Add(name);
            return Values.Count - 1;
        }

        public object getValue(int index)
        {
            if (index < 0 || index >= Values.Count)
                return null;
            return Values[index];
        }

        public object getValue(string name)
        {
            for (int i = 0; i < Names.Count; i++)
            {
                if (Names[i].ToString() == name)
                    return Values[i];
            }
            return null;
        }

        public void getValueRef(string name, out object obj)
        {
            for (int i = 0; i < Names.Count; i++)
            {
                if (Names[i].ToString() == name)
                    obj = Values[i];
            }
            obj = null;
        }

        public bool setValue(string name, object obj)
        {
            for (int i = 0; i < Names.Count; i++)
            {
                if (Names[i].ToString() == name)
                {
                    Values[i] = obj;
                    return true;
                }
            }
            return false;
        }

        public bool setValue(string name, object obj, string polymorphType)
        {
            for (int i = 0; i < Names.Count; i++)
            {
                if (Convert.ToString(Names[i]) == name)
                {
                    try
                    {
                        Union u = (Union)Values[i];
                        u.CurrentValue = obj;
                        u.CurrentType = polymorphType;
                    }
                    catch
                    {
                        try
                        {
                            Any a = (Any)Values[i];
                            a.CurrentValue = obj;
                            a.CurrentType = polymorphType;
                        }
                        catch { Values[i] = obj; }
                    }
                    return true;
                }
            }
            return false;
        }

        public bool setValue(int index, object obj)
        {
            if (index < 0 || index >= Values.Count)
                return false;

            Values[index] = obj;
            return true;
        }

        public override string ToString()
        {
            StringBuilder sb = new StringBuilder("[");
            for (int i = 0; i < TypeData.Count; i++)
            {
                if (i > 0 && (i < TypeData.Count))
                    sb.Append(", ");
                String s = TypeData[i].ToString();
                /*string, float64, bool, int64, Tuple, View*/
                switch (s)
                {
                    case "string":
                        sb.Append("String " + Names[i].ToString() + ": \"" +
Values[i].ToString() + "\"");
                        break;
                    case "float64":
                        sb.Append("Float " + Names[i].ToString() + ": " +
Values[i].ToString());
                        break;
                    case "int64":
                        sb.Append("Integer " + Names[i].ToString() + ": " +
Values[i].ToString());
                        break;
                    case "bool":
                        sb.Append("Boolean " + Names[i].ToString() + ": " +
Values[i].ToString());
                        break;
                    case "Tuple":
                        sb.Append("Tuple " + Names[i].ToString() + ": " +
Values[i].ToString());
                        break;
                    case "View":
                        sb.Append("View " + Names[i].ToString() + ": " +
Values[i].ToString());
                        break;
                    case "Union":
                        sb.Append("Union " + Names[i].ToString() + ": " +
Values[i].ToString());
                        break;
                    case "Any":
                        sb.Append("Any " + Names[i].ToString() + ": " +
Values[i].ToString());
                        break;
                    case "Sequence":
                        sb.Append("Sequence " + Names[i].ToString() + ": " +
Values[i].ToString());
                        break;
                    case "Connection":
                        sb.Append("Connection " + Names[i].ToString() + ": " +
Values[i].ToString());
                        break;
                }
            } //end for
            sb.Append("]");
            return sb.ToString();
        } //end ToString

        public void deepCopy(View v)
        {
            for (int i = 0; i < TypeData.Count; i++)
            {
                String s = TypeData[i].ToString();
                /*string, float64, bool, int64, Tuple, View*/
                switch (s)
                {
                    case "string":
                        Values[i] = v.Values[i].ToString();
                        break;
                    case "float64":
                        double dcd = (Double)v.Values[i];
                        Values[i] = dcd;
                        break;
                    case "int64":
                        long dcl = Convert.ToInt64(v.Values[i]);
                        Values[i] = dcl;
                        break;
                    case "bool":
                        bool dcb = (Boolean)v.Values[i];
                        Values[i] = dcb;
                        break;
                    case "Tuple":
                        Tuple tt = (Tuple)Values[i];
                        tt.deepCopy((Tuple)v.Values[i]);
                        break;
                    case "View":
                        View vt = (View)Values[i];
                        vt.deepCopy((View)v.Values[i]);
                        break;
                    case "Union":
                        Union ut = (Union)Values[i];
                        ut.deepCopy((Union)v.Values[i]);
                        break;
                    case "Any":
                        Any at = (Any)Values[i];
                        at.deepCopy((Any)v.Values[i]);
                        break;
                    case "Sequence":
                        Sequence st = (Sequence)Values[i];
                        st.deepCopy((Sequence)v.Values[i]);
                        break;
                    case "Connection":
                        Values[i] = v.Values[i];
                        break; //do nothing
                } //end switch
            } //end for
        } //end deepCopy

        public void deepClone(View v)
        {
            for (int i = 0; i < v.TypeData.Count; i++)
            {
                String s = v.TypeData[i].ToString();
                TypeData.Add(s);
                Names.Add(v.Names[i].ToString());
                /*string, float64, bool, int64, Tuple, View*/
                switch (s)
                {
                    case "string":
                        Values.Add(v.Values[i].ToString());
                        break;
                    case "float64":
                        double dcd = (Double)v.Values[i];
                        Values.Add(dcd);
                        break;
                    case "int64":
                        long dcl = Convert.ToInt64(v.Values[i]);
                        Values.Add(dcl);
                        break;
                    case "bool":
                        bool dcb = (Boolean)v.Values[i];
                        Values.Add(dcb);
                        break;
                    case "Tuple":
                        Tuple tt = new Tuple();
                        tt.deepClone((Tuple)v.Values[i]);
                        Values.Add(tt);
                        break;
                    case "View":
                        View vt = new View();
                        vt.deepClone((View)v.Values[i]);
                        Values.Add(vt);
                        break;
                    case "Union":
                        Union ut = new Union();
                        ut.deepClone((Union)v.Values[i]);
                        Values.Add(ut);
                        break;
                    case "Any":
                        Any at = new Any();
                        at.deepClone((Any)v.Values[i]);
                        Values.Add(at);
                        break;
                    case "Sequence":
                        Sequence st = new Sequence();
                        st.deepClone((Sequence)v.Values[i]);
                        Values.Add(st);
                        break;
                    case "Connection":
                        Values.Add(v.Values[i]);
                        break;
                } //end switch
            } //end for
        } //end deepClone

        public override bool deepCompare(PiConstructedType ct)
        {
            View v;
            try
            {
                v = (View)ct;
            }
            catch { return false; }

            if (this.TypeData.Count != v.TypeData.Count)
                return false;

            for (int i = 0; i < v.Values.Count; i++)
            {
                if (this.TypeData[i].ToString() != v.TypeData[i].ToString())
                    return false;

                if (isConstructedType(this.TypeData[i].ToString()))
                {
                    PiConstructedType oct = (PiConstructedType)this.Values[i];
                    PiConstructedType pct = (PiConstructedType)v.Values[i];
                    if (!pct.deepCompare(oct))
                        return false;
                }

                if (TypeData[i].ToString() == "Connection")
                {
                    AbstractConnection oc = (AbstractConnection)Values[i];
                    AbstractConnection pc = (AbstractConnection)v.Values[i];
                    if (!oc.IsEquivalent(pc))
                        return false;
                }
            }
            return true;
        } //end deepCompare

        public bool fillViewFromConsole(String prefix)
        {
            Console.Out.WriteLine(prefix + "Enter view data:");
            String input = String.Empty;

            for (int i = 0; i < TypeData.Count; i++)
            {
                try
                {
                    switch (TypeData[i].ToString())
                    {
                        case "string":
                            Console.Out.Write(prefix + "String " +
Names[i].ToString() + ": ");
                            input = Console.In.ReadLine();
                            Values[i] = input;
                            break;
                        case "float64":
                            Console.Out.Write(prefix + "Float " +
Names[i].ToString() + ": ");
                            input = Console.In.ReadLine();
                            Values[i] = Convert.ToDouble(input);
                            break;
                        case "int64":
                            Console.Out.Write(prefix + "Integer " +
Names[i].ToString() + ": ");
                            input = Console.In.ReadLine();
                            Values[i] = Convert.ToInt64(input);
                            break;
                        case "bool":
                            Console.Out.Write(prefix + "Boolean " +
Names[i].ToString() + ": ");
                            input = Console.In.ReadLine();
                            Values[i] = Convert.ToBoolean(input);
                            break;
                        case "Tuple":
                            Tuple tt = (Tuple)Values[i];
                            tt.fillTupleFromConsole(prefix + "  ");
                            Values[i] = tt;
                            break;
                        case "View":
                            View vt = (View)Values[i];
                            vt.fillViewFromConsole(prefix + "  ");
                            Values[i] = vt;
                            break;
                        case "Union":
                            Union ut = (Union)Values[i];
                            ut.fillUnionFromConsole(prefix + "  ");
                            Values[i] = ut;
                            break;
                        case "Any":
                            Any at = (Any)Values[i];
                            at.fillAnyFromConsole(prefix + "  ");
                            Values[i] = at;
                            break;
                        case "Sequence":
                            Sequence st = (Sequence)Values[i];
                            st.fillSequenceFromConsole(prefix + "  ");
                            Values[i] = st;
                            break;
                        case "Connection":
                            break; //do nothing
                    } //end switch
                }
                catch
                {
                    Console.Out.WriteLine("Input is not valid. Please try again.");
                    i--;
                }
            } //end for
            return true;
        } //end fillViewFromConsole

    } //end View
```

II.3 Polymorph

```
/*********************************************************************************
*********************************************************************************
Class Polymorph: This class is parent to the Union and Any classes and provides
methods common to both data types.
*********************************************************************************
*********************************************************************************/

    public class Polymorph : PiConstructedType
    {
        public object CurrentValue; //any base or constructed type
        public string CurrentType;

        public Polymorph()
        {
        }
        public bool isTypeEqual(Polymorph p)
        {
            if (p.CurrentType == CurrentType)
            {
                if (isConstructedType(CurrentType))
                {
                    PiConstructedType ct = (PiConstructedType)CurrentValue;
                    PiConstructedType st = (PiConstructedType)p.CurrentValue;
                    return ct.deepCompare(st);
                }

                if (p.CurrentType == "Connection")
                {
                    AbstractConnection ct = (AbstractConnection)CurrentValue;
                    AbstractConnection st = (AbstractConnection)p.CurrentValue;
                    return ct.IsEquivalent(st);
                }
                return true;
            }
            return false;
        } //end isTypeEqual
    }
```

II.4 Union

```
/***********************************************************************************
***********************************************************************************
                                    Class Union
***********************************************************************************
**********************************************************************************/

    public class Union : Polymorph
    {
        public ArrayList TypeData; //always strings
        public ArrayList TypeStructures; //empty strings for basic types, class
objects for constructed types

        public Union()
        {
            TypeData = new ArrayList();
            TypeStructures = new ArrayList();
            CurrentType = "";
            CurrentValue = "";
        }

        public int addType(string type, object structure)
        {
            TypeData.Add(type);
            TypeStructures.Add(structure);
            return TypeStructures.Count - 1;
        }

        public bool setValue(string type, object val)
        {
            for (int i = 0; i < TypeData.Count; i++)
            {
                string dType = TypeData[i].ToString();
                if (dType == type)
                {
                    if (isConstructedType(type))
                    {
                        PiConstructedType ct = (PiConstructedType)val;
                        PiConstructedType st =
(PiConstructedType)TypeStructures[i];
                        if (!ct.deepCompare(st))
                        continue;
                    }

                    if (type == "Connection")
                    {
                        AbstractConnection ct = (AbstractConnection)val;
                        AbstractConnection st =
(AbstractConnection)TypeStructures[i];
                        if (!ct.IsEquivalent(st))
                            continue;
                    }

                    CurrentValue = val;
                    CurrentType = type;
                    return true;
                }
            }
            return false;
        } //end setValue

        public bool setValue(object val)
        {
            CurrentValue = val;
            return true;
        } //end setValue

        public override bool deepCompare(PiConstructedType ct)
        {
            Union u;
            try
            {
                u = (Union)ct;
            }
            catch { return false; }

            if (this.TypeData.Count != u.TypeData.Count)
                return false;

            for (int i = 0; i < u.TypeData.Count; i++)
            {
                if (this.TypeData[i].ToString() != u.TypeData[i].ToString())
                    return false;

                if (isConstructedType(this.TypeData[i].ToString()))
                {
                    PiConstructedType oct =
(PiConstructedType)this.TypeStructures[i];
                    PiConstructedType pct = (PiConstructedType)u.TypeStructures[i];
                    if (!pct.deepCompare(oct))
                        return false;
                }

                if (TypeData[i].ToString() == "Connection")
                {
                    AbstractConnection oc = (AbstractConnection)TypeStructures[i];
                    AbstractConnection pc =
(AbstractConnection)u.TypeStructures[i];
                    if (!oc.IsEquivalent(pc))
                        return false;
                }
            }
            return true;
        } //end deepCompare

        public void deepCopy(Union u)
        {
            CurrentType = u.CurrentType;
            if (u.CurrentType == "Connection")
            {
                CurrentValue = u.CurrentValue;
                return;
            }

            if (isConstructedType(u.CurrentType))
            {
                switch (CurrentType)
                {
                    case "Tuple":
                        Tuple tt = new Tuple();
                        tt.deepClone((Tuple)u.CurrentValue);
                        CurrentValue = tt;
                        break;
                    case "View":
                        View vt = new View();
                        vt.deepClone((View)u.CurrentValue);
                        CurrentValue = vt;
                        break;
                    case "Union":
                        Union ut = new Union();
                        ut.deepClone((Union)u.CurrentValue);
                        CurrentValue = ut;
                        break;
                    case "Any":
                        Any at = new Any();
                        at.deepClone((Any)u.CurrentValue);
                        CurrentValue = at;
                        break;
                    case "Sequence":
                        Sequence st = new Sequence();
                        st.deepClone((Sequence)u.CurrentValue);
                        CurrentValue = st;
                        break;
                }
            }
            else
            {
                CurrentValue = u.CurrentValue;
            }
        } //end deepCopy

        public void deepClone(Union u)
        {
            deepCopy(u);
            for (int i = 0; i < u.TypeData.Count; i++)
            {
                String s = u.TypeData[i].ToString();
                TypeData.Add(s);
                /*string, float64, bool, int64, Tuple, View, Union*/
                switch (s)
                {
                    case "string":
                    case "float64":
                    case "int64":
                    case "bool":
                        TypeStructures.Add("");
                        break;
                    case "Tuple":
                        Tuple tt = new Tuple();
                        tt.deepClone((Tuple)u.TypeStructures[i]);
                        TypeStructures.Add(tt);
                        break;
                    case "View":
                        View vt = new View();
                        vt.deepClone((View)u.TypeStructures[i]);
                        TypeStructures.Add(vt);
                        break;
                    case "Union":
                        Union ut = new Union();
                        ut.deepClone((Union)u.TypeStructures[i]);
                        TypeStructures.Add(ut);
                        break;
                    case "Any":
                        Any at = new Any();
                        at.deepClone((Any)u.TypeStructures[i]);
                        TypeStructures.Add(at);
                        break;
                    case "Sequence":
                        Sequence st = new Sequence();
                        st.deepClone((Sequence)u.TypeStructures[i]);
                        TypeStructures.Add(st);
                        break;
                    case "Connection":
                        Object connO;
                        AbstractConnection c =
(AbstractConnection)u.TypeStructures[i];
                        c.deepClone(out connO);
                        TypeStructures.Add(c);
                        break;
                } //end switch
            } //end for
        } //end deepClone

        public override string ToString()
        {
            StringBuilder sb = new StringBuilder("[");
            for (int i = 0; i < TypeData.Count; i++)
            {
                if (i > 0 && (i < TypeData.Count))
                    sb.Append(", ");
                String s = TypeData[i].ToString();
                sb.Append(NetToPiADLTypeName(s));

                if (s == CurrentType)
                {
                    if (isConstructedType(s))
                    {
                        PiConstructedType c1 =
(PiConstructedType)TypeStructures[i];
                        PiConstructedType c2 = (PiConstructedType)CurrentValue;
                        if (c1.deepCompare(c2))
                            sb.Append("*: " + CurrentValue.ToString());
                    }
                    else if (CurrentType == "string")
                        sb.Append("*: \"" + CurrentValue.ToString() + "\"");
                    else
                        sb.Append("*: " + CurrentValue.ToString());
                }
                else if (isConstructedType(s))
                    sb.Append(": " + TypeStructures[i].ToString());

            } //end for
            sb.Append("]");
            return sb.ToString();
        } //ToString

        public bool fillUnionFromConsole(String prefix)
        {
            //First give a numbered list of the set of available types
            //Then verify the validity of the input and store it in the union
            Console.Out.WriteLine(prefix +
"Specify type by selecting a number and press Enter:");
            String input = String.Empty;
            int typeIndex = 0;
            while (typeIndex == 0)
            {
                //numbered list
                for (int j = 0; j < TypeData.Count; j++)
                {
                    int i = j + 1;
                    switch (TypeData[j].ToString())
                    {
                        case "bool":
                            Console.Out.Write(prefix + " " + i + " - Boolean\n");
                            break;
                        case "int64":
                            Console.Out.Write(prefix + " " + i + " - Integer\n");
                            break;
                        case "float64":
                            Console.Out.Write(prefix + " " + i + " - Float\n");
                            break;
                        case "string":
                            Console.Out.Write(prefix + " " + i + " - String\n");
                            break;
                        case "Tuple":
                            Tuple tt = (Tuple)TypeStructures[j];
                            Console.Out.Write(prefix + " " + i + " - Tuple " +
tt.ToString() + "\n");
                            break;
                        case "View":
                            View vt = (View)TypeStructures[j];
                            Console.Out.Write(prefix + " " + i + " - View " +
vt.ToString() + "\n");
                            break;
                        case "Union":
                            Union ut = (Union)TypeStructures[j];
                            Console.Out.Write(prefix + " " + i + " - Union " +
ut.ToString() + "\n");
                            break;
                        case "Any":
                            Any at = (Any)TypeStructures[j];
                            Console.Out.Write(prefix + " " + i + " - Any " +
at.ToString() + "\n");
                            break;
                        case "Sequence":
                            Sequence st = (Sequence)TypeStructures[j];
                            Console.Out.Write(prefix + " " + i + " - Sequence " +
st.ToString() + "\n");
                            break;
                        case "Connection":
                            AbstractConnection ct =
(AbstractConnection)TypeStructures[j];
                            Console.Out.Write(prefix + " " + i + " - Connection " +
ct.ToString() + "\n");
                            break;
                    } //end switch
                } //end for
                try
                {
                    typeIndex = Convert.ToInt32(Console.In.ReadLine());
                }
                catch
                {
                    Console.Out.Write("Invalid input. Please try again.");
                    typeIndex = 0;
                }
            }
            typeIndex--;
            bool inputInvalid = true;
            //Now take input according to the type selected
            while (inputInvalid)
            {
                inputInvalid = false;
                try
                {
                    CurrentType = TypeData[typeIndex].ToString();
                    switch (CurrentType)
                    {
                        case "string":
                            Console.Out.Write(prefix + "String: ");
                            input = Console.In.ReadLine();
                            CurrentValue = input;
                            break;
                        case "float64":
                            Console.Out.Write(prefix + "Float: ");
                            input = Console.In.ReadLine();
                            CurrentValue = Convert.ToDouble(input);
                            break;
                        case "int64":
                            Console.Out.Write(prefix + "Integer: ");
                            input = Console.In.ReadLine();
                            CurrentValue = Convert.ToInt64(input);
                            break;
                        case "bool":
                            Console.Out.Write(prefix + "Boolean: ");
                            input = Console.In.ReadLine();
                            CurrentValue = Convert.ToBoolean(input);
                            break;
                        case "Tuple":
                            Tuple tt = (Tuple)TypeStructures[typeIndex];
                            tt.fillTupleFromConsole(prefix + "  ");
                            CurrentValue = tt;
                            break;
                        case "View":
                            View vt = (View)TypeStructures[typeIndex];
                            vt.fillViewFromConsole(prefix + "  ");
                            CurrentValue = vt;
                            break;
                        case "Union":
                            Union ut = (Union)TypeStructures[typeIndex];
                            ut.fillUnionFromConsole(prefix + "  ");
                            CurrentValue = ut;
                            break;
                        case "Any":
                            Any at = (Any)TypeStructures[typeIndex];
                            at.fillAnyFromConsole(prefix + "  ");
                            CurrentValue = at;
                            break;
                        case "Sequence":
                            Sequence st = (Sequence)TypeStructures[typeIndex];
                            st.fillSequenceFromConsole(prefix + "  ");
                            CurrentValue = st;
                            break;
                        case "Connection":
                            AbstractConnection ct =
(AbstractConnection)TypeStructures[typeIndex];
                            object ctconn;
                            ct.deepClone(out ctconn);
                            CurrentValue = ctconn;
                            break;
                    } //end switch
                }
                catch
                {
                    Console.Out.WriteLine("Input is not valid. Please try again.");
                    inputInvalid = true;
                }
            }
            return true;
        } //end fillUnionFromConsole
```

```
    } //end Union
```

II.5 Sequence

```
/*********************************************************************************
**********************************************************************************
                                  Class Sequence
**********************************************************************************
*********************************************************************************/

    public class Sequence : PiConstructedType
    {
        public ArrayList Values;
        public string Type;
        public object Structure;

        public Sequence()
        {
            Values = new ArrayList();
            Type = "";
            Structure = "";
        }

        public void setType(string _type, object _structure)
        {
            Type = _type;
            Structure = _structure;
        }

        public void addValue(object val)
        {
            Values.Add(val);
        }

        public void setValue(object val, long index)
        {
            if (Values.Count < index + 1)
            {
                for (int i = Values.Count - 1; i < index; i++)
                {
                    switch (Type)
                    {
                        case "string":
                            string oString = "";
                            Values.Add(oString);
                            break;
                        case "float64":
                            double oDouble = 0.0;
                            Values.Add(oDouble);
                            break;
                        case "int64":
                            long oLong = 0;
                            Values.Add(oLong);
                            break;
                        case "bool":
                            bool oBool = false;
                            Values.Add(oBool);
                            break;
                        case "Tuple":
                            Tuple tt = new Tuple();
                            tt.deepClone((Tuple)Structure);
                            Values.Add(tt);
                    break;
                case "View":
                    View vt = new View();
                    vt.deepClone((View)Structure);
                    Values.Add(vt);
                    break;
                case "Union":
                    Union ut = new Union();
                    ut.deepClone((Union)Structure);
                    Values.Add(ut);
                    break;
                case "Any":
                    Any at = new Any();
                    at.deepClone((Any)Structure);
                    Values.Add(at);
                    break;
                case "Connection":
                    Object connO;
                    AbstractConnection c = (AbstractConnection)Structure;
                    c.deepClone(out connO);
                    Values.Add(connO);
                    break;
                case "Sequence":
                    Sequence st = new Sequence();
                    st.deepClone((Sequence)Structure);
                    Values.Add(st);
                    break;
            } //end switch
        } //end for
    } //end if
    Values[(int)index] = val;
} //end setValue

public void setValue(string type, object val, long index) //for union and
                                                           //any
{
    if (Values.Count < index + 1)
    {
        for (int i = Values.Count - 1; i < index; i++)
        {
            switch (Type)
            {
                case "Union":
                    Union ut = new Union();
                    ut.deepClone((Union)Structure);
                    Values.Add(ut);
                    break;
                case "Any":
                    Any at = new Any();
                    at.deepClone((Any)Structure);
                    Values.Add(at);
                    break;
            } //end switch
        } //end for
    } //end if
    if (Type == "Union")
    {
        Union au = (Union)Values[(int)index];
        au.setValue(type, val);
    }
    if (Type == "Any")
    {
        Any aa = (Any)Values[(int)index];
        aa.setValue(type, val);
    }
} //end setValue

public object getValue(long index)
```

```
{
    if (Values.Count < index + 1)
    {
        for (int i = Values.Count - 1; i < index; i++)
        {
            switch (Type)
            {
                case "string":
                    string oString = "";
                    Values.Add(oString);
                    break;
                case "float64":
                    double oDouble = 0.0;
                    Values.Add(oDouble);
                    break;
                case "int64":
                    long oLong = 0;
                    Values.Add(oLong);
                    break;
                case "bool":
                    bool oBool = false;
                    Values.Add(oBool);
                    break;
                case "Tuple":
                    Tuple tt = new Tuple();
                    tt.deepClone((Tuple)Structure);
                    Values.Add(tt);
                    break;
                case "View":
                    View vt = new View();
                    vt.deepClone((View)Structure);
                    Values.Add(vt);
                    break;
                case "Union":
                    Union ut = new Union();
                    ut.deepClone((Union)Structure);
                    Values.Add(ut);
                    break;
                case "Any":
                    Any at = new Any();
                    at.deepClone((Any)Structure);
                    Values.Add(at);
                    break;
                case "Connection":
                    Object connO;
                    AbstractConnection c = (AbstractConnection)Structure;
                    c.deepClone(out connO);
                    Values.Add(connO);
                    break;
                case "Sequence":
                    Sequence st = new Sequence();
                    st.deepClone((Sequence)Structure);
                    Values.Add(st);
                    break;
            } //end switch
        } //end for
    } //end if
    return Values[(int)index];
} //end setValue

public override string ToString()
{
    StringBuilder output = new StringBuilder();
    output.Append("Sequence[" + NetToPiADLTypeName(Type) + "] : (");
    int count = 0;
    foreach (object o in Values)
    {
        if (count > 0)
```

```
                output.Append(", ");
            output.Append(o.ToString());
            count++;
        }
        output.Append(")");
        return output.ToString();
    }

    public void deepCopy(Sequence s)
    {
        Values.Clear();
        foreach (Object o in s.Values)
        {
            switch (s.Type)
            {
                case "string":
                    string oString = (string)o;
                    Values.Add(oString);
                    break;
                case "float64":
                    double oDouble = (double)o;
                    Values.Add(oDouble);
                    break;
                case "int64":
                    long oLong = Convert.ToInt64(o);
                    Values.Add(oLong);
                    break;
                case "bool":
                    bool oBool = (bool)o;
                    Values.Add(oBool);
                    break;
                case "Tuple":
                    Tuple tt = new Tuple();
                    tt.deepClone((Tuple)o);
                    Values.Add(tt);
                    break;
                case "View":
                    View vt = new View();
                    vt.deepClone((View)o);
                    Values.Add(vt);
                    break;
                case "Union":
                    Union ut = new Union();
                    ut.deepClone((Union)o);
                    Values.Add(ut);
                    break;
                case "Any":
                    Any at = new Any();
                    at.deepClone((Any)o);
                    Values.Add(at);
                    break;
                case "Connection":
                    Values.Add(o);
                    break;
                case "Sequence":
                    Sequence st = new Sequence();
                    st.deepClone((Sequence)o);
                    Values.Add(st);
                    break;
            }//end switch
        } //end foreach
    } //end deepCopy

    public void deepClone(Sequence s)
    {
        Type = s.Type;
        switch (s.Type)
        {
                case "string":
                case "float64":
                case "int64":
                case "bool":
                    Structure = "";
                    break;
                case "Tuple":
                    Tuple tt = new Tuple();
                    tt.deepClone((Tuple)s.Structure);
                    Structure = tt;
                    break;
                case "View":
                    View vt = new View();
                    vt.deepClone((View)s.Structure);
                    Structure = vt;
                    break;
                case "Union":
                    Union ut = new Union();
                    ut.deepClone((Union)s.Structure);
                    Structure = ut;
                    break;
                case "Any":
                    Any at = new Any();
                    at.deepClone((Any)s.Structure);
                    Structure = at;
                    break;
                case "Connection":
                    Object connO;
                    AbstractConnection c = (AbstractConnection)s.Structure;
                    c.deepClone(out connO);
                    Structure = connO;
                    break;
                case "Sequence":
                    Sequence st = new Sequence();
                    st.deepClone((Sequence)s.Structure);
                    Structure = st;
                    break;
            } //end switch

            deepCopy(s);
        }//end deepClone

        public bool fillSequenceFromConsole(String prefix)
        {
            Console.Write(prefix +
"Please specify the number of members in the sequence: ");
            string input; bool inpValid = false; int length = 0;
            while (!inpValid)
            {
                input = Console.In.ReadLine();
                try
                {
                    length = Convert.ToInt32(input);
                    inpValid = true;
                }
                catch
                {
                    Console.Out.WriteLine(prefix +
"Input is not valid. Please try again.");
                }
            } //end while

            Values.Clear();
            for (int i = 0; i < length; i++)
            {
                try
                {
                    switch (Type)
```

```
                        {
                            case "string":
                                Console.Out.Write(prefix + "String: ");
                                input = Console.In.ReadLine();
                                Values.Add(input);
                                break;
                            case "float64":
                                Console.Out.Write(prefix + "Float: ");
                                input = Console.In.ReadLine();
                                Values.Add(Convert.ToDouble(input));
                                break;
                            case "int64":
                                Console.Out.Write(prefix + "Integer: ");
                                input = Console.In.ReadLine();
                                Values.Add(Convert.ToInt64(input));
                                break;
                            case "bool":
                                Console.Out.Write(prefix + "Boolean: ");
                                input = Console.In.ReadLine();
                                Values.Add(Convert.ToBoolean(input));
                                break;
                            case "Tuple":
                                Tuple tt = new Tuple();
                                tt.deepClone((Tuple)Structure);
                                tt.fillTupleFromConsole(prefix + "  ");
                                Values.Add(tt);
                                break;
                            case "View":
                                View vt = new View();
                                vt.deepClone((View)Structure);
                                vt.fillViewFromConsole(prefix + "  ");
                                Values.Add(vt);
                                break;
                            case "Union":
                                Union ut = new Union();
                                ut.deepClone((Union)Structure);
                                ut.fillUnionFromConsole(prefix + "  ");
                                Values.Add(ut);
                                break;
                            case "Any":
                                Any at = new Any();
                                at.deepClone((Any)Structure);
                                at.fillAnyFromConsole(prefix + "  ");
                                Values.Add(at);
                                break;
                            case "Sequence":
                                Sequence st = new Sequence();
                                st.deepClone((Sequence)Structure);
                                st.fillSequenceFromConsole(prefix + "  ");
                                Values.Add(st);
                                break;
                            case "Connection":
                                AbstractConnection ct = (AbstractConnection)Structure;
                                object oct;
                                ct.deepClone(out oct);
                                Values.Add(oct);
                                break;
                        } //end switch
                    }
                    catch
                    {
                        Console.Out.WriteLine(prefix +
"Input is not valid. Please try again.");
                        i--;
                    }
                }
                return false;
            } //end fillSequenceFromConsole
```

```
        public override bool deepCompare(PiConstructedType ct)
        {
            Sequence s;
            try
            {
                s = (Sequence)ct;
            }
            catch { return false; }

            if (Type != s.Type)
                return false;

            if (isConstructedType(Type))
            {
                if
(!((PiConstructedType)s.Structure).deepCompare((PiConstructedType)Structure))
                    return false;
            }
            return true;
        } //end deepCompare

    }//end class Sequence
```

Appendix III: π-ADL Model of an SMPP Gateway

```
Actor names behaviour
{

      params : view [phoneNumber : String, simID : String, recipientList : String,
            SMSText : String];
      paramConn : Connection [ view [phoneNumber : String, simID : String,
            recipientList : String, SMSText : String] ];
      ReceptionServParams : view[operation : String, data : any];
      outConn : Connection [view[operation : String, data : any]];
      resultConn : Connection [view[operation : String, data : any]];
      result : view[operation : String, data : any];

      via out send "\n------- ACTOR starts ----";

      via GetUserData send Void where {paramConn renames dataConn};
      via paramConn receive params;

      ReceptionServParams::operation = "sendSMS";
      ReceptionServParams::data = params;


      compose
      {
            via out send "\n ----- Actor: SEND ReceptionServParams --";
            via outConn send ReceptionServParams;
      and
            via out send "\n ----- Actor: SEND MobileCapabilities --";
            via MobileCapabilities send Void where {outConn renames inConnectionC,
                  resultConn renames outConnectionC};
      and
            via out send "\n ----- Actor: waiting to RECEIVE result --";
            via resultConn receive result;

            via out send "\n ======= RESULT: ";
            via out send result::data;
      and
            done;
      }
} //end Actor

value GetUserData is abstraction ()
{
      data : view [phoneNumber : String, simID : String, recipientList : String,
            SMSText : String];
      dataConn : Connection [ view [phoneNumber : String, simID : String,
            recipientList: String, SMSText : String] ];

      via out send "\n------- GetUserData starts --";

      data::phoneNumber = "666";
      data::simID = "5555";
      data::recipientList = "MMMM - ZZZZ - FFFF";
      data::SMSText = "Message Text";

      compose
      {
            via dataConn send data;
      and
            done;
      }
} //end GetUserData
```

```
// CONTRACT -----> MessagePattern: Query/Response
value MobileCapabilities is abstraction ()
{
// Connections used to communicate with the services
       inConnectionS : Connection [view[operation : String, data : any]];
       outConnectionS : Connection [view[operation : String, data : any]];

       inConnectionC : Connection [view[operation : String, data : any]];
       outConnectionC : Connection [view[operation : String, data : any]];

// Information token received/sent
       inData : view [operation : String, data : any];

// Message exchange pattern which reflects the state machine
       input: view[numStates: Integer,
       MEP : sequence [view [state_id: Integer, via_SERVID : String,
              op : String, numNextStates: Integer,
              next : sequence[view[criteria: String, newState: Integer]]]],
              numConn: Integer,
              ServConnGroup : sequence [view[SERVID : String,
              inServConn : Connection[view[operation : String, data : any]],
              outServConn : Connection[view[operation : String, data : any]]]]];

       MessagePattern : sequence [view [state_id: Integer, via_SERVID : String,
              op : String, numNextStates: Integer,
              next : sequence[view[criteria: String,
              newState:Integer]]]];
       state : view [state_id: Integer, via_SERVID : String,
              op : String, numNextStates: Integer,
              next : sequence[view[criteria: String, newState:Integer]]];

       nextState: view[criteria: String, newState:Integer];

       maxcountState : Integer;
       maxCountNewState : Integer;
       countState : Integer;
       countNewState : Integer;

       currentState : Integer;

       ServConnGroup : sequence [view[SERVID : String,
              inServConn : Connection[view[operation : String, data : any]],
              outServConn : Connection[view[operation : String, data : any]]]];

       OneServConn :view[SERVID : String,
              inServConn : Connection[view[operation : String, data : any]],
              outServConn : Connection[view[operation : String, data : any]]];
       S : String;

       via out send "\n <><>< Contract MobileCapabilities starts ><><>";

       OneServConn::SERVID = "ReceptionService";
       OneServConn::inServConn = inConnectionS ;
       OneServConn::outServConn = outConnectionS ;

       ServConnGroup(0) = OneServConn;

// Description of the Message Exchange Protocol
// Message Exchange Pattern: Query/Response

       MessagePattern(0)::state_id = 0;
       MessagePattern(0)::via_SERVID = "C";
       MessagePattern(0)::op = "receive";
       MessagePattern(0)::numNextStates = 1;
       MessagePattern(0)::next(0)::criteria = "sendSMS";
       MessagePattern(0)::next(0)::newState = 1;

       MessagePattern(1)::state_id = 1;
       MessagePattern(1)::via_SERVID = "S";
       MessagePattern(1)::op = "send";
       MessagePattern(1)::numNextStates = 1;
       MessagePattern(1)::next(0)::criteria = "sendSMS";
       MessagePattern(1)::next(0)::newState = 2;

       MessagePattern(2)::state_id = 2;
       MessagePattern(2)::via_SERVID = "S";
       MessagePattern(2)::op = "receive";
       MessagePattern(2)::numNextStates = 1;
       MessagePattern(2)::next(0)::criteria = "sendSMS";
       MessagePattern(2)::next(0)::newState = 3;

       MessagePattern(3)::state_id = 3;
       MessagePattern(3)::via_SERVID = "C";
       MessagePattern(3)::op = "send";
       MessagePattern(3)::numNextStates = 1;
       MessagePattern(3)::next(0)::criteria = "sendSMS";
       MessagePattern(3)::next(0)::newState = 0;

       input::MEP = MessagePattern;
       input::ServConnGroup = ServConnGroup;
       input::numStates = 4;

       MessagePattern = input::MEP;
       maxcountState = input::numStates;
       inConnectionS = input::ServConnGroup(0)::inServConn;
       outConnectionS = input::ServConnGroup(0)::outServConn;
       S = input::ServConnGroup(0)::SERVID;

       currentState = 0;
       while (currentState > -1) do
       {
         countState = 0;
         while (countState < maxcountState) do
         {

           state = MessagePattern(countState);
           countState = countState + 1;
           if (currentState == state::state_id) do
           {
             via out send "\n\n ..... Current State: ";
             via out send currentState;
             via out send "\n ..... DATA: ";
             via out send inData;

             if (state::op == "receive") do // receive
             {
               if (state::via_SERVID == "S") do
               {
                 via out send "\n ~~~~~ waiting RECEIVE SERVER ~~~~~";
                 via inConnectionS receive inData;
               }
             else do
             {
                 via out send "\n ~~~~~ waiting RECEIVE CLIENT ~~~~~";
                 via inConnectionC receive inData;
             }
           }
           else do
           {
             // To send anything to the server a compose should be done:
             // first to send the data through the connection
             // second to execute the abstraction and unify the connections
             if (state::via_SERVID == "S") do{
               via out send "\n ~~~~~ waiting SEND SERVER ~~~~~";
               compose
               {
```

```
                    via outConnectionS send inData;
                  and
                    via dynamic(S) send Void where {outConnectionS renames inConn,
                       inConnectionS renames outConn};
                  }
               }
               else do
               {
                  via out send "\n ~~~~~ waiting SEND CLIENT ~~~~~";
                  via outConnectionC send inData;
               }
            }

            countNewState = 0;
            maxCountNewState = state::numNextStates;
            while (countNewState < maxCountNewState) do
            {
               nextState = state::next(countNewState);
               countNewState = countNewState + 1;

               if (nextState::criteria == inData::operation) do
               {
                  via out send "\n >>>>> New state ==> ";
                  via out send nextState::newState;
                  currentState = nextState::newState;
               }
            }
          }
        }
    }
} //end MobileCapabilities

//+++++++++++++++++++++++++++++++ CONTRACT <------

// -------------------------------------------------- RECEPTION SUBSYSTEM
// -------------------------- RECEPTION SERVICE
value ReceptionService is abstraction ()
{
      outConn : Connection[view [operation : String, data : any]];
      output : view [operation : String, data : any];

      inConn : Connection[view [operation : String, data : any]];
      input : view [operation : String, data : any];

      resultConn : Connection [String];
      result : String;

      via out send "\n\n +++++ RECEPTION SERVICE starts";

      via inConn receive input;
      via out send "\n +++++ Reception INPUT: ";
      via out send input;

      if (input::operation == "sendSMS") do
      {
            //via SendSMS send input::data where {resultConn renames resultConn};
            //via resultConn receive result;
            compose
            {
                  result = "test";
                  output::operation = "sendSMS";
                  output::data = result;

                  via out send "\n +++ Reception Service: send output +++ \n";
                  via outConn send output;
            and
                  done;
            }
```

```
        }
} //end ReceptionService

behaviour
{
        shConnection : Connection[view[operation : String, data : any]];

        via out send "\n----- main BEHAVIOUR starts --";
        compose
        {
                via SMSCenter send shConnection;
        and
                via processingSubsystem send shConnection;
        }
} //end behaviour

value SMSCenter is abstraction (outConn : Connection[view[operation : String,
                data : any]])
{
        output : view[operation : String, data : any];
        end : Boolean;

        via out send "\n <<<<< SMSCenter: starts <<<<<";

        end = false;
        while (!end) do
        {
                via out send "\n <<<<< SMSCenter: waiting in RECEIVE <<<<<";
                via outConn receive output;

                unobservable; // perform inner operation with the data received
                via out send "\n <<<<< SMSCenter: Data received = ";
                via out send output::data;
        }
} //end SMSCenter

// -------------------------------------------------- PROCESSING SUBSYSTEM
value processingSubsystem is abstraction (startInfo : Connection[view[operation :
                String, data : any]])
{
        connSet : view[connDS : Connection[view[operation : String, data : any]],
                connSMSCenter: Connection[view[operation : String, data : any]]];
        shConnDS : Connection[view[operation : String, data : any]];


        via out send "\n ----- PROCESSING SUBSYSTEM starts ------------";
        connSet::connSMSCenter = startInfo;
        connSet::connDS = shConnDS;
        compose
        {
           via DirectoryService send connSet;
         and
           via SMSSenderService send shConnDS;
         }
} //end processingSubsystem

value SMSSenderService is abstraction (connDS : Connection[view[operation : String,
                data : any]])
{
         querySMSCenter : view[operation : String, data : any];
        queryDS : view[operation : String, data : any];
        answerDS : view[operation : String, data : view [contractConn :
                Connection[view[operation : String, data : any]],
                servMetaInfo : any]];
        answerDSAny : view[operation : String, data : any];
        clientDataResponse : view [contractConn :
                Connection[view[operation : String, data : any]], servMetaInfo : any];
        via out send "\n ->->- SMSSender starts ->->->->->";

        queryDS::operation = "locateService";
        queryDS::data = "service criteria";

        via out send "\n ->-> SMSSender: awaiting SEND (Directory query) ->->";
        via connDS send queryDS;

        via out send "\n ->-> SMSSender: awaiting RECEIVE (Directory answer) ->->";
        via connDS receive answerDSAny;

          select answerDSAny::data
        {
                case view [contractConn : Connection[view[operation : String,
                                data : any]], servMetaInfo : any] do
                        answerDS::data = answerDSAny::data;
          }

        answerDS::operation = answerDSAny::operation;

// answerDS is a view of operation (value: "locateService") and another view
// (containing the connection to the contract and metainfo about the service

        via out send "\n ->->- SMSSender: analyzing DIRECTORY answer->->->->->";
        clientDataResponse::contractConn = answerDS::data::contractConn;
        clientDataResponse::servMetaInfo = answerDS::data::servMetaInfo;

        compose
        {
                querySMSCenter::operation = "sendSMS";
                querySMSCenter::data = "SMS No.1";
                via out send "\n ->->- SMSSender: proceeding to send SMS No.1 ->->->";
                via clientDataResponse::contractConn send querySMSCenter;

                querySMSCenter::operation = "sendSMS";
                querySMSCenter::data = "SMS No.2";
                via out send "\n ->->- SMSSender: proceeding to send SMS No.2 ->->->";
                via clientDataResponse::contractConn send querySMSCenter;
        and
                done;
        }
} //end SMSSenderService

value DirectoryService is abstraction (connSet : view[connDS :
                Connection[view[operation : String, data : any]],
                connSMSCenter: Connection[view[operation : String, data : any]]])
{
        input : view[operation : String, data : any];
        inConn : Connection[view[operation : String, data : any]];
        locationConn : view[inConn : Connection[view[operation : String,
                data : any]], input : view[operation : String, data : any],
                connSMSCenter: Connection[view[operation : String, data : any]]];
        end : Boolean;

        via out send "\n ~^~^~ Directory starts ~^~^~";

        locationConn::connSMSCenter = connSet::connSMSCenter;
        inConn = connSet::connDS;

        end = false;
        while (!end) do
        {
          via out send "\n ~^~ Directory: waiting to receive REQUEST ~^~";
          via inConn receive input;

// send back who has sent me (inconn), what (input) and what should be
//answered (connSMSCenter) through the locateService abstraction
          locationConn::inConn = inConn;
          locationConn::input = input;

          via out send "\n ~^~ Directory: redirect query to locating operation ~^~";
          if (input::operation == "locateService") do
            via locateService send locationConn;
        }
}

value locateService is abstraction (locationConn : view[inConn :
              Connection[view[operation : String, data : any]],
              input : view[operation : String, data : any],
              connSMSCenter: Connection[view[operation : String, data : any]]])
{
        shContractClientConn : Connection [view [operation : String, data : any]];
        contractStartInfo : view [MEP : any,
              connServ1 : Connection[view[operation : String, data : any]],
              connServ2 : Connection[view[operation : String, data : any]]];
        clientDataResponse : view [contractConn : Connection[view[operation :
              String, data : any]], servMetaInfo : any];
        clientResponse : view [operation : String, data : any];
        requestData : any;

        via out send "\n ++-++ LOCATOR in action ++-++-++-++-++";

        requestData = locationConn::input::data;
        unobservable; //analyze request data
        compose
        {
// Prepare and create connector: Shipping
              contractStartInfo::MEP = "";     //full description of the
                                               //message exchange pattern
              contractStartInfo::connServ1 = locationConn::connSMSCenter;
              contractStartInfo::connServ2 = shContractClientConn;

              via out send "\n ++-++ LOCATOR: Shipping created ++-++-++-++-++";
              via Shipping send contractStartInfo;
        and
              clientDataResponse::contractConn = shContractClientConn;
              clientDataResponse::servMetaInfo = ""; //info about the service
                                          //operations and semantics of the service

              clientResponse::data = clientDataResponse;
              clientResponse::operation = "locateService";

              via out send "\n ++-++ LOCATOR: answer to query SENT ++-++-++-++-++";
              via locationConn::inConn send clientResponse;
        and
              done;
        }
} //end DirectoryService

value Shipping is abstraction (locationConn : view[MEP : any,
              connServ1 : Connection[view[operation : String, data : any]],
              connServ2 : Connection[view[operation : String, data : any]]])
{
// Connections used to communicate with the services
        inConnectionS : Connection [view[operation : String, data : any]];
        inConnectionC : Connection [view[operation : String, data : any]];
// Information token received/sent
        inData : view [operation : String, data : any];

        end : Boolean;

        inConnectionS = locationConn::connServ1;
        inConnectionC = locationConn::connServ2;

        via out send "\n <><>< SHIPPING starts <><><><><>";
        end = false;
        while (!end) do
        {
                via out send "\n <><>< Shipping: waiting client RECEIVE <><><><><>";
                via inConnectionC receive inData;
                unobservable;
                via out send "\n <><>< Shipping: waiting server SEND <><><><><>";
                via inConnectionS send inData;
        }
} //end Shipping
```

References

[Acm08] The AcmeStudio Home Page. Retrieved in February 2008 from http://www.cs.cmu.edu/~acme/AcmeStudio/index.html.

[AG94] R. Allen, D. Garlan, “Formalizing Architectural Connection,” 16th International Conference on Software Engineering, Sorrento, Italy, May 1994.

[AG98] M. Abadi, A. D. Gordon, "A Calculus for Cryptographic Protocols: The Spi Calculus," Fourth ACM Conference on Computer and Communications Security, 1998.

[Ald03] J. Aldrich, “Using Types to Enforce Architectural Structure,” University of Washington Ph.D. Dissertation, August 2003.

[All97] Robert J. Allen, “A Formal Approach to Software Architecture”, Ph.D. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144, May, 1997.

[And08] The Andro MDA website. Retrieved in September 2008 from http://www.andromda.org.

[APR05] N. Ali, J. Perez, I. Ramos, J.A. Carsi, "Introducing Ambient Calculus in Mobile Aspect-Oriented Software," 5th Working IEEE/IFIP Conference on Software Architecture, 2005.

[BJC05] T. Batista, A. Joolia and G. Coulson, “Managing Dynamic Reconfiguration in Component-Based Systems,” Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005.

[BKR04] N. Benton, A. Kennedy, and C. Russo, “Adventures in interoperability: the SML.NET experience,” Proceedings of the 6th ACM SIGPLAN International conference on Principles and Practice of Declarative Programming, Vol.6, 2004.

[Bla07] R. Blackwell, The L Sharp.NET website. Retrieved in November 2007 from http://www.lsharp.org/.

[Bme08] The B Method website. Retrieved in December 2008 from http://www.bmethod.org.

[BPS06] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler and F. Yergeau, “Extensible Markup Language (XML) 1.0 - Fourth Edition,” W3C Recommendation, August 2006.

[Bro87] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," Computer, Vol. 20, No. 4 (April 1987).

[Bro90] R. A. Brooks, “Elephants Don't Play Chess,” Robotics and Autonomous Systems 6, 1990.

[Bur02] K. Burton, “.NET Common Language Runtime Unleashed,” SAMS Publishing, ISBN 0-672-32124-6, Indianapolis IN, USA, Apr 2002.

[BW05] F.R.M. Barnes and P.H. Welch, “Communicating Mobile Processes, Lecture Notes in Computer Science,” Vol. 3525, 2005.

[CFB02] C. E. Cuesta, P. Fuente, M. Barrio-Solórzano and E. Beato, “Coordination in a Reflective Architecture Description Language,” Springer Berlin / Heidelberg Lecture Notes in Computer Science, 2002.

[CLM05] S. Carpineti, C. Laneve, and P. Milazzo, “BoPi – a distributed machine for experimenting Web Services technologies,” Fifth International Conference on Application of Concurrency to System Design, 2005.

[CMV03] P. Cáceres, E. Marcos and B. Vela, “A MDA-Based Approach for Web Information System Development,” Workshop in Software Model Engineering, San Francisco, USA, 2003.

[CO03] V. Cremet and M. Odersky, “PiLib: A Hosted Language for Pi-Calculus Style Concurrency,” EPFL Dagstuhl Proceedings: Domain-Specific Program Generation, 2003.

[CT06] J. Cabot, E. Teniente, "Constraint support in MDA tools: A survey", European Conference on Model Driven Architecture No2, Bilbao, Spain 2006.

[Dij72] E. W. Dijkstra, “The Humble Programmer,” ACM Turing Lecture, 1972.

[DVT01] E. M. Dashofy, A. Van der Hoek and R. N. Taylor, “A Highly-Extensible, XML-Based Architecture Description Language,” Proceedings of the Working IEEE/IFIP

Conference on Software Architectures, Amsterdam Netherlands, 2001.

[Ecl08] The Eclipse Foundation website. Retrieved in October 2008 from http://www.eclipse.org.

[Fal04] D. C. Fallside ed., “XML Schema Part 0: Primer,” W3C Recommendation, October 2004.

[FDH04] J. Fanchon, K. Drira, S.P. Hernandez, "Abstract channels as connectors for software components in group communication services," Proceedings of the Fifth Mexican International Conference in Computer Science, 2004.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom, “Exploiting Style in Architectural Design Environments,” Proceedings of SIGSOFT’94: Foundations of Software Engineering, December 1994.

[GMW97] D. Garlan, R. Monroe and D. Wile, “ACME: An Architecture Description Interchange Language,” Proceedings of CASCON’97, November 1997.

[GP95] D. Garland, D. Perry, “Introduction to Special issue on software architecture,” IEEE Transactions on Software Engineering, April 1995.

[GQ94] M. Gorlick, A. Quilici, “Visual programming-in-the-large versus visual programming-in-the-small,” Proceedings of IEEE Symposium on Visual Languages, 1994.

[GR91] M. M. Gorlick and R. R. Razouk, “Using Weaves for Software Construction and Analysis,” Proceedings of the 13th International Conference on Software Engineering (ICSE13), pages 23-34, Austin, TX, May 1991.

[Gra08] The Graphviz website. Retrieved in June 2008 from http://www.graphviz.org.

[GS03] J. Greenfield, K. Short, “Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools,” 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2003.

[HS02] B. Hailpern and P. Santhanam, “Software debugging, testing and verification,” IBM Systems Journal of Software Testing and Verification, 2002. 41(1): pp. 3–14.

[Hoa85] C. A. R. Hoare, “Communicating Sequential Processes”, Prentice Hall, Englewood Califfs, NJ, 1985.

[Hic07] R. Hickey, The DotLisp website. Retrieved in November 2007 from http://dotlisp.sourceforge.net/dotlisp.htm.

[IEE00] IEEE Standard 1516-2000, IEEE standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules.

[IEE00a] “Recommended practice for architectural description of software-intensive systems,” ANSII/IEEE Std 1471-2000.

[JBC05] A. Joolia, T. Batista, G. Coulson, A.T.A. Gomes, "Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform," 5th Working IEEE/IFIP Conference on Software Architecture, Pittsburgh, PA USA, 2005.

[Jer97] M. Jern, “Information Drill-down using Web tools,” First International Conference on Information Visualisation (IV'97), 1997.

[KGO01] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic and R.N. Taylor, “xADL: Enabling Architecture-Centric Tool Integration With XML,” 34th Hawaii International Conference on System Sciences (HICSS-34), Maui, Hawaii, January, 2001.

[KVL98] J.W. Krueger, S. Vestal and B. Lewis, "Fitting the Pieces Together: Systems/Software Analysis and Code Integration using MetaH," Proceedings of the 17th Digital Avionics Systems Conference Proceedings, 1998.

[LCV00] B. Lewis, E. Colbert and S. Vestal. “Developing Evolvable, Embedded, Time-Critical Systems with MetaH,” Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00), 2000.

[Lid02] S. Lidin, “Inside Microsoft .NET IL Assembler,” Microsoft Press, ISBN 0-7356-1547-0, Redmond WA, USA, 2002.

[LKA95] D.C. Luckham, J. Kenney , L.M. Augustin, J. Vera, D. Bryan, W. Andmann, “Specification and analysis of system architecture using rapide,” IEEE Trans. Softw. Eng. Vol. 21, No. 4, 1995.

[LKJ07] I. Loulou , A. H. Kacem, M. Jmaiel and K. Drira, "Formal Design of Structural and Dynamic Features of Publish/Subscribe Architectural Styles," Springer Berlin /

Heidelberg Lecture Notes in Computer Science, Book: Software Architecture, September 2007

[LL99] N. Levy, F. Losavio, “Analyzing and comparing architectural styles,” XIX International Conference of the Chilean Computer Science Society, 1999.

[LLR04] F. Losavio1 , N. Levy and A. Ramdane-Cherif, "Architectural Design for a Wireless Environment," Springer Berlin / Heidelberg Lecture Notes in Computer Science, Book: Embedded and Ubiquitous Computing, July 2004

[Low05] J. Lowy, “Programming .NET Components 2nd Edition Appendix D: Generics,” ISBN: 978-0596102074, O'Reilly Media Inc., 2005.

[Luc96] D. C. Luckham, "Rapide: A language and toolset for simulation of distributed systems by partial orderings of events," Princeton University, 1996.

[LV95] D.C. Luckham and J. Vera, “An event-based architecture definition language,” IEEE Trans.Softw. Eng. Vol. 21, No. 9, pp. 717–734, Sept 1995.

[LVM95] D. C. Luckham, J. Vera and S. Meldal, “Three Concepts of a System Architecture,” Stanford University, 1995.

[LWF03] A. Lopes, M. Wermelinger, and J. L. Fiadeiro, “Higher-order architectural connectors,” ACM Transactions in Software Engineering. Methodology. 2003.

[MB02] S. Mellor, M. Balcer, “Executable UML: A foundation for model-driven architecture,” Addison Wesley, 2002.

[MDE95] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, “Specifying Distributed Software Architectures,” Proceedings of the Fifth European Software Engineering Conference, September 1995.

[Met08] MetaH Language and Tools. Retrieved in February 2008 from http://www.htc.honeywell.com/metah/tools.html.

[Mil93] R. Milner, “The Polyadic π-Calculus: A Tutorial, Logic and Algebra of Specification”, Springer-Verlag, 1993.

[Mil99] R. Milner, Communicating and Mobile Systems: The π-Calculus, Cambridge University Press, 1999.

[MKS89] J. Magee, J. Kramer and M. Sloman, “Constructing Distributed Systems in Conic,” IEEE Transactions on Software Engineering, 1989. 15(6): pp. 663-675.

[MMO05] D. Manset, R. McClatchey, F. Oquendo and H. Verjus, “A Model-Driven Approach for Grid Services Engineering,” Eighteenth International Conference on Software and Systems Engineering and their Applications (ICSSEA'05), Paris, France, Dec 2005.

[Mon08] The Mono Project Website. Retrieved in February 2008 from http://www.mono-project.com.

[MO06] R. Mateescu and F. Oquendo, “π-AAL: an architecture analysis language for formally specifying and verifying structural and behavioural properties of software architectures,” SIGSOFT Software Engineering Notes, March 2006.

[MOR96] N. Medvidovic, P Oreizy, JE Robbins, RN Taylor, “Using object-oriented typing to support architectural design in the C2 style,” SIGSOFT’96 - San Francisco, CA, October, 1996.

[MR99] N. Medvidovic and D.S. Rosenblum, “Assessing the Suitability of a Standard Design Method for Modeling Software Architectures,” Proceedings of the First IFIP Working Conference on Software Architecture, pp. 161-182, WICSA1, San Antonio, TX, Feb 1999.

[MRR02] N. Medvidovic, D.S. Rosenblum and D.F. Redmiles, “Modeling Software Architectures in the Unified Modeling Language,” ACM Trans. Software Eng. And Methodology, Vol. 11, No. 1, pp. 2-57, Jan 2002.

[MSD07] Type Class, Microsoft Online MSDN Library. Retrieved in September 2007 from http://msdn.microsoft.com/en-us/library/system.type.aspx.

[MSD08] AutoResetEvent Class (System.Threading), Microsoft online MSDN Library. Retrieved in February 2008 from http://msdn2.microsoft.com/en-us/library/system.threading.autoresetevent.aspx.

[MSD08a] Interlocked Class, Microsoft online MSDN Library. Retrieved in February 2008 from http://msdn2.microsoft.com/en-us/library/system.threading.interlocked(VS.71).aspx.

[MT00] N. Medvidovic and R.N. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages,” IEEE Trans. Software Eng., Vol. 26, No. 1, Jan. 2000.

[MT96] N. Medvidovic and R. N. Taylor, “Reusing Off-the-Shelf Components to Develop a Family of Applications in the C2 Architectural Style,” Proceedings of the First International Workshop on Software Architectures for Product Families (IW-SAPF-1), Las Navas del Marques, Avila, Spain, November, 1996.

[MTR97] R. Milner, M. Tofte and R. Harper, “The Definition of Standard ML (Revised),” MIT Press, 1997.

[Nig04] E. G. Nigles, “Build Your Own .NET Language and Compiler”, Apress, ISBN: 1-59059-134-8, New York, NY, USA, 2004.

[OAS06] OASIS, “Reference Model for Service Oriented Architecture,” Committee draft 1.0, 2006. Retrieved in March 2008 from http://www.oasis-open.org/committees/download.php/ 16587/wd-soa-rm-cd1ED.pdf.

[OMG01] OMG, “Model Driven Architecture,” Eds.: J. Miller, J. Mukerji, Document No. ormsc/2001-07-01.

[Oqu04] F. Oquendo, “π-ADL: An Architecture Description Language based on the Higher Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures,” ACM Software Engineering Notes, No. 3, May 2004.

[Oqu04a] F. Oquendo, “π-ARL: An Architecture Refinement Language for Formally Modelling the Stepwise Refinement of Software Architectures,” ACM Software Engineering Notes, Vol. 29, Issue 5, Sept 2004.

[Oqu05] F. Oquendo, “Tutorial on ArchWare ADL – Version 2 (π-ADL Tutorial),” ArchWare European RTD Project IST-2001-32360, Jun 2005.

[Oqu06] F. Oquendo, “Formally Modelling Software Architectures with the UML 2.0 Profile for π-ADL,” ACM Software Engineering Notes, Vol. 31, No. 1, Jan 2006.

[OWM04] F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, C. Occhipinti, “ArchWare: Architecting Evolvable Software,” Proceedings of the 1st European Workshop on Software Architecture, LNCS 3047, Springer Verlag, May 2004.

[PW92] D. E. Perry and A. L. Wolf, “Foundations for the Study of Software Architecture,” ACM SIGSOFT Software Engineering Notes, 1992.

[QDO06] Z. Qayyum, W. Di, F. Oquendo, “π-ADL Visual Notation and its Application to Formally Modeling the High Level Architecture,” 19th International Conference on Software and Systems Engineering and their Applications, 2006.

[RMQ95] R. Riemenschneider, M. Moriconi and X. Qian, “Correct Architecture Refinement,” IEEE Transactions on Software Engineering, vol. 21, no. 4, pages 356-372, April 1995.

[RSS01] A. Regev, W. Silverman, E. Shapiro, “Representation and Simulation of Biochemical Processes Using the pi-Calculus Process Algebra,” Pacific Symposium on Biocomputing, 2001.

[SC06] M. Shaw, P. Clements, “The golden age of software architecture,” IEEE Software Volume 23, Issue 2, March-April 2006 Page(s): 31 – 39.

[Sch01] S. Schneider, "The B-method," Palgrave, 2001. ISBN 033379284X, 9780333792841.

[SDK95] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young and G. Zelesnik, “Abstractions for Software Architecture and Tools to Support Them,” IEEE Transactions on Software Engineering, vol. 21, no. 4, pages 314-335, April 1995.

[SM07] D. Syme, J. Margetson, The F# website. Retrieved in November 2007 from http://research.microsoft.com/fsharp/.

[SMP07] SMPP Forum, “SMPP v5.0 Specification”. Retrieved in December 2007 from http://www.smsforum.net/.

[Spe00] J. Spencer, ed. “Architecture Description Markup Language (ADML): Creating an Open Market for IT Architecture Tools,” Open Group White Paper, September, 2000.

[Spy08] Altova GmbH, XML Spy Software. Retrieved in October 2008 from http://www.xmlspy.com.

[STA08] B. Suleiman, V. Tosic and E. Aliev, "Non-functional property specifications for

WRIGHT ADL," 8th IEEE International Conference on Computer and Information Technology, 2008.

[Sut05] H. Sutter, “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software,” Dr. Dobb's Journal, March 2005

[Ves98] S. Vestal, "MetaH User's Manual, 1.27 ed.," Honeywell Technology Center Minneapolis, USA, 1998.

[Wik07] Recursive Descent Parser - Wikipedia, Retrieved in December 2007 from http://en.wikipedia.org/wiki/Recursive_descent_parser.

[WWW07] Validator for XML Schema, World Wide Web Consortium. Retrieved in December 2007 from http://www.w3.org/2000/09/webdata/xsv.

Abbreviations

ADL	Architecture Description Language
BNF	Backus-Naur Form
BSD	Berkeley System Distribution
CIL	Common Intermediate Language
CIM	Computation Independent Model
CLR	Common Language Runtime
CSP	Communicating Sequential Processes
DBMS	Database Management System
DSL	Domain Specific Language
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework
HLA	High Level Architecture
MDA	Model Driven Architecture
ML	Metalanguage
OMG	Object Management Group
OO	Object Oriented
PIM	Platform Independent Model
PSM	Platform Specific Model
SDK	Software Development Kit
SDLC	Software Development Life Cycle
SMPP	Simple Message Passing Protocol
SMS	Short Message Service
SMSC	Short Message Service Center
SOA	Service Oriented Architecture
SOC	Service Oriented Computing
UML	Unified Modeling Language

Résumé

L'architecture logicielle est devenue un thème scientifique majeur de l'informatique. En effet, l'architecture logicielle fournit l'abstraction qui permet de développer rigoureusement et de faire évoluer des systèmes logiciels complexes au vu des besoins tant fonctionnels que non fonctionnels.

Afin de modéliser les architectures logicielles, un nouveau type de langage est apparu : les langages de description d'architectures (ADL, Architecture Description Language).

Divers ADL ont été proposés dans la littérature, mais ces ADL sont restreints à la modélisation d'architectures abstraites, indépendantes des plateformes d'implémentation. Lors de l'implémentation, l'architecture n'est plus représentée.

Cette thèse s'inscrit dans le domaine des ADL et porte sur la définition et la mise en œuvre d'un langage pour la concrétisation, c'est-à-dire l'implémentation explicite, d'architectures logicielles.
Elle adresse le problème de la construction d'un tel langage et son système d'exécution. Pour cela elle aborde le problème d'un point de vue nouveau : la construction d'un langage centré sur l'architecture logicielle.

Assis sur des bases formelles, notamment sur le π-calcul et π-ADL, ces travaux ont donné lieu à un langage formel pour décrire et garantir l'intégrité architecturale d'un système au niveau de sa spécification, de son implémentation et de ses évolutions ultérieures. La machine virtuelle et le compilateur associé sont enfouis dans la plateforme .NET.