Dynamic Distributed Systems Design: An Architectural Design and Verification Approach

by
Anna Liu

Thesis submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy
at the
School of Computer Science and Engineering of the
University of New South Wales
Sydney, NSW 2052, Australia

1998

Abstract

With the availability of fast network technology and powerful desktop computers, there is an increasing demand for the construction of reliable software that can exploit the parallel processing power of a network of computers. These typical distributed applications include those commonly found in telecommunication and banking industries. To support the simpler development of these complex software systems, robust middleware technologies such as CORBA, DCE and ActiveX/COM are becoming widely deployed. Middleware provides a higher level programming interface for distributed software engineers, as the low-level networking details have been hidden.

However, writing distributed software is still a complex task. Although the low-level distribution issues can be mostly ignored with the assistance of middleware, fundamental distributed system issues such as concurrency, synchronisation, dynamic process creation and deletion and re-configurable communication structures still need to be carefully considered.

This thesis presents a novel software architecture design and verification methodology. Architects employ a pragmatic, graphical design method called Dynamic PARSE (PARSE-D) to design the software architecture. At the same time, they capture the concurrent and dynamic features of the system. Such dynamic features include the creation and deletion of processes and re-configurable communication links. Lastly, the correctness of the design can be verified, and possible design faults may be detected by using an automatic design analysis and verification tool called PARSE-DAT.

Acknowledgments

Three years is not a short time. There have been many people that made this possible, and truly deserve my utmost gratitude.

Firstly, my supervisor Dr. Ian Gorton, for his guidance throughout the past three years. He has encouraged me to explore different avenues, as well as pointing out the way at confusing times. He has been a great inspiration in many ways, a great mentor and a great friend.

Secondly, my co-supervisor Dr. John Zic, for his assistance in the area of concurrency formalisms and his constant encouragement and support.

Thirdly, Dr. Innes Jelly from Sheffield Hallam University, for her guidance and feedback on my work throughout the duration of my thesis work.

The π-Calculus discussion group at the University of New South Wales and Macquarie University has been a fruitful experience, thanks to its participants: Dr. John Zic, Dr. John Potter, Dr. Arcot Sowmya, Dr. Ken Robinson, Dr. Axel Wabenhorst, Ms. Winnie Qun, Mr. Xiaogang Zhang, and Mr. Partha Roop.

Many other computer scientists also provided useful feedback at various stages of this work: Dr. Jonathan Gray, Dr. Peter Croll, Dr. Carl Birkinshaw, Dr. Stefano Russo, Mr Alan Kinnersly, and Mr Patrick Jordan for proofreading the hardcopy.

Many Ph.D. students and research scientists at UNSW shared their valuable personal research experiences as Ph.D. candidates and provided mutual encouragement. Thanks to Mr. Swee Yew Choe, Dr. Ricky Chan, Dr. Alvin Chan, Mr. Joseph Kam, Dr. Herbert Chan, Dr. Jinsong Ouyoung, Mr. Tommy Cheung, Mr. Lei Hu and Mr. Lei Miao.

Also many thanks to Dr. Bjorn Victor for providing the model checking tool Mobility Workbench.

Lastly, but not the least, thanks to my parents and close friends who provided the vital support behind the scenes. It is their sacrifices that made this thesis possible.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation and Goals

1.1.1 Why Build Distributed Software

There are many reasons for developing distributed software.

System entities are often identified as being distributed at the user analysis level (or in the problem domain). For example, in a banking application, typical entities include customers, transactions, tellers, and data stores. These entities are usually located at different places. In this situation, distributed applications model the problem domain naturally.

Collaborative work environment is an application area where it is natural to model both the problem domain and solution domain in a distributed manner. In this situation, where users are located at different places, accessing various resources either locally or remotely, it is natural to have an implementation that reflects this distributed nature, and resources such as data stores can be located, duplicated, or migrated closer to the client application to increase system efficiency. This is only possible with a distributed implementation.

Hence, and perhaps more importantly, distributed software exploits the power of networks of computing resources. These distributed applications use spare resources located at different locations within the network [Gorton95a].

Second, with the availability of symmetric multiprocessing machines in the desktop computer market, applications can be developed to utilise this increased processing power. Applications built using a multithreaded approach will benefit from improved performance, lower cost, and higher reliability.

Lastly, distributed teams developing different software components is an effective and efficient way of developing large systems. As long as the interface between the distributed components is well-defined, engineers can work independently, possibly at different physical locations and in different time-zones. Various system components can be implemented by different experts from different areas, and engineers in different teams need only to focus on the implementation of their allocated component.

1.1.2 Difficulties of Writing Distributed Software

Despite all the benefits of constructing distributed software, there are many difficulties involved in this process. In addition to the usual difficulties in writing sequential programs, other problems must be dealt with.

Implementing distributed systems requires low-level communication support. The inter-process communication across a network is complex. If the programmer has to manage all the detailed networking operations while programming a high-level application, errors are likely to be introduced [Kramer94]. This problem is further complicated if the underlying network is heterogeneous.

It is also essential that large, distributed systems are constructed using sound software engineering practices that promote scaleability and modularity, maximise reusability and attempt to guarantee reliability and correctness to some degree. Ideally, such a software development methodology for distributed systems should be easy for software engineers to learn and use; it should provide formal verification to validate the software correctness; and facilitate automatic program generation from the system design [Gorton97a].

Specifically, one major difficulty relating to the lack of software engineering methodology for distributed systems is inadequacy of design notations. There is no appropriate design notation that caters for evolving systems with dynamic features. Such features include dynamic process creation and deletion, and communication reconfiguration. However, the architecture of many existing systems change during execution - for example, systems built using CORBA and OLE. These dynamic properties which are prevalent in many distributed systems have a large impact on the correctness of the software under construction [Liu95].

Secondly, existing design methodologies (e.g. object-oriented design methods) do not place enough importance on the various concurrency issues [Low96]. Such issues include synchronization and global controls. Phenomena such as deadlocks, livelocks and race conditions need to be carefully considered when designing a distributed system.

In close relation to the previous point regarding concurrency issues, there is a lack of analysis and verification tools for distributed software. The analysis of possible undesirable properties such as deadlocks, livelocks, and race conditions, the verification of specification/design conformance, and performance prediction are important issues to consider. However, these analysis tasks are difficult, and thus the development of support tools for distributed software analysis is important.

Further on the issue of tool support for design analysis, the state explosion phenomenon is one problem associated with model checking tools. The state explosion phenomenon occurs when the search space increases exponentially [Peterson91]. Many heuristics such as semantic minimization [Elseaidy96], localization reduction [Kurshan94] and the use of partial order information [Peled96] are promising approaches that alleviate the state explosion problem.

It is important to bridge the gap between these formal approaches and practicing software engineers. The incorporation of formal methods into a software engineering process is just as important as the development of the formal methods and tools.

1.1.3 Attempts to Overcome These Difficulties

The various attempts to overcome the above mentioned difficulties are discussed here.

There are various distributed operating systems that aim to manage distributed resources at the operating systems level. Such distributed operating systems include Mach [Rashid89] and Chorus [Rozier90]. Common desktop operating systems such as Windows NT [Richter94] [Custer93] and Solaris [Sun98] provide a rich set of facilities to enable applications to exploit multiple processors. However, the application programming interface support for inter-process communication across a network is minimal and low-level (and hence error prone).

The emerging middleware technology provides an abstraction for the underlying communication required by distributed components, simplifying the implementation process. Middleware has thus created a strong interest in the software community in building distributed systems. Available middleware products include the Object Management Group's CORBA [OMG95], and Microsoft's DCOM [Microsoft98]. These systems enable the construction of extensible software with reusable, distributed components.

The field of software architecture aims to provide support for the design of the architecture of large systems, and advocates the importance of not only data structures and functionality, but also inter-component communication synchronisation issues. Recent work in this area has demonstrated some of the failings of widely used development methodologies for distributed systems [Shaw96]. For example, problems have been highlighted with object-oriented design methods for distributed system design, despite the fact that the distributed software engineer faces various complex issues relating to the partitioning and distribution of software components [Magee95], existing object-oriented approaches do not have provisions for architectural issues. There is still much work to be done in the software architecture area, especially in the area of dynamic architectures.

A well established research community exists in the area of concurrency formalism. Such work include CSP [Hoare85], CCS [Milner89], π-Calculus [Milner91] and Petri Nets [Peterson81]. These formalisms were designed to assist in the modeling of concurrent systems, and through using mathematically sound foundations, can be used to perform system analysis and verification against some particular properties/criteria. Some examples here include observational equivalence check using CCS [Milner80] and reachability analysis using Petri Nets [Peterson81]. However, these formalisms are complex and difficult for software engineers to learn and use [Saiedian96]. In addition, the use of such formalisms for a large-scale software system is expensive, yet at the same time, does not guarantee any extra correctness/performance benefits, over conventional testing methods, largely due to the lack of tool support and non-scaleable techniques [Holloway96].

1.1.4 Concluding Remark

What is lacking in the field of distributed software engineering is an overall software engineering method that supports the modeling of dynamic software architecture at both the analysis and design level, and also aids the effective employment of formalisms for certain critical sections of the system. This methodology should be supported by a user-friendly CASE tool which facilitates the editing of the software architecture model, allows multiple views of the system model, supports navigation, and automates design analysis and verification.

1.2 Contribution

This thesis presents a software architecture design methodology for dynamic distributed systems called Dynamic PARSE (PARSE-D). PARSE-D provides an explicit representation of the parallelism and distribution in a system via a well defined set of model elements. It also enables the designer to verify their design at an early stage of the engineering process, hence reducing resource overheads incurred in discovering deadlocks at the later testing stage. There is also a supporting tool-set called PARSE-DAT that aids design construction and automates formal analysis and verification.

The PARSE-D work extends the PARSE project [Gorton95] to cater for the design of loosely coupled distributed applications. So far, the focus of the work with PARSE has been upon tightly coupled, high-performance parallel software systems which are implemented in parallel programming languages such as Occam, Ada or Parallel C. PARSE-D deals with more coarse grain distributed systems typically implemented with conventional programming languages such as C, C++, or Java, with concurrency and inter-process communication facilities provided either by an underlying distributed operating system, such as Windows NT or UNIX, or middleware such as CORBA, Active X/DCOM, and DCE.

The following is a summary of the contributions made by this thesis.

- The extension to an architecture description language called PARSE, in order to incorporate dynamic features. The resultant set of notations is known as the Dynamic PARSE Process Graph Design Notation, which has the extra features of creation and deletion of process components, and dynamic communication reconfiguration.
- The definition of formal semantics of the Dynamic PARSE Process Graph Design Notation. The two chosen formalisms are π-Calculus [Milner91] and a variant of Petri-Net [Peterson81] called Self-Modifying Nets [Valk77].
- A distributed software engineering methodology (PARSE-D) supporting design using Dynamic PARSE Process Graph Notation and design analysis/verification utilising π-Calculus.
- An integrated tool-set environment called PARSE-DAT for supporting the PARSE-D software engineering methodology. The two major components are the CASE tool PARSE-DT which enables the construction of Dynamic PARSE designs; and an automated analysis/verification tool called PARSE-AT.

1.3 Thesis outline

- Chapter 2 presents a literature review. This focuses on existing software architecture design methods, and also discusses current distributed computing environments and concurrency formalisms.
- Chapter 3 presents the Dynamic PARSE Design methodology.
- Chapter 4 presents the Dynamic PARSE Verification Methodology.

- Chapter 5 discusses the implementation of the graphical editing environment PARSE-DT, and the automated analysis/verification environment PARSE-AT.
- Chapter 6 presents a number of case studies. These case studies range from the simple client-server architecture, pipeline architecture, to more complex applications of a communication protocol and a collaborative work environment.
- Chapter 7 concludes this thesis, and presents possible directions for further work.

2. Review

This chapter presents a literature review of existing work on supports for the engineering of distributed systems. The three major areas of work to be covered are:

- software architecture methodologies
- formal methods for architectural design verification
- distributed computing environments

Each of the above research areas will be examined in turn, and major projects in each area identified and evaluated.

2.1 Software Architecture Methodologies

Research in the area of software architecture aims to provide support for the design of the architecture of large systems [Shaw96]. Typically, these systems consist of various software components, and span a network of computing resources. The design of software architecture is thus a level of design concerned with the specification of the overall system structure, where these structural issues include:

- general component organization,
- global control structure,
- protocols for communication,
- synchronization,
- data access,
- assignment of functionality to design elements,
- physical distribution,
- composition of design elements,
- scaling and performance,
- selection among design alternatives.

The low level implementation issues such as algorithms and data structures are not considered in the software architecture design, and are delayed till a later stage of the engineering process [Shaw96].

At the centre of software architecture research is the development of architectural definition languages (ADLs). These are well-defined languages that facilitate the description of an architecture's components and connections. The languages are usually graphical, and provide some form of *box and line* syntax for specifying components and hooking them together. These ADLs also have been formerly known as Module Interconnection Languages MILs [Rice94].

There are a number of software architecture methodologies, each at varying degrees of maturity. Most of these software architecture methodologies have a basis in graphical ADLs: Darwin [Magee95], LOTOS [Leduc94], SDL [Faergemand91] and PARSE [Gorton95], while others provide textual ADLs: C2 [Taylor96] and Rapide [Luckham95], and some also have formal foundations based on well defined mathematical constructs: SDL, LOTOS and Estelle [Diaz89]. Similarly, the development processes associated with the methodologies are at varying levels of detail, some are rather primitive and cover no more than a simple instruction on how to use the corresponding ADLs; while others are much more refined and powerful, and may incorporate external formal analysis methods (e.g. UniCon [Shaw96], PARSE). Various supporting tools have also been built, such as the Software Architect's Assistant [Ng95] for Darwin and Aesop [Garland94] [Monroe96] for the ABLE project. These tools typically provide a visual rapid prototyping and modeling environment. Some provide simulation of events: C2, some provide code generation: Occam generator for PARSE [Gorton96b], and some allow for external design analysis: UniCon or performance evaluation: HL for PARSE [Hu97].

ABLE

Carnegie Mellon University's ABLE project is concerned with exploring and developing the concept of Architectural Style, and building tools that practicing software architects might find useful [Monroe96][Garland94]. The tool development effort has focused on the Aesop system, a toolkit for rapidly producing software architecture design and analysis environments that are customized to support various specific architectural styles. Aesop has three main features:

- a generic object model for representing architectural designs.
- the characterisation of architectural styles as specialisation of this object model (through sub-typing).
- a toolkit for creating an open architectural design environment from a description of a specific architectural style.

The architectural styles mentioned here include the commonly found architectural patterns and idioms such as client-server, pipeline and filter, layered systems, and blackboards. Additional examples include localised ones such as model-view-controller and various object-oriented patterns [Gamma95], as well as various reference models for communication such as CORBA [OMG95] and OSI [Spragins92], and user-interface frameworks.

UniCon

UniCon (Universal Connector Language) is an ADL providing representations for components and connectors, abstractions and encapsulation, types and type checking. It can also describe a system which involves coordinating real-time tasks. An example is a system with two 'schedulable' tasks that interact through remote procedure calls. UniCon researchers employ rate-monotonic analysis (RMA) techniques to analyse the real-time properties of the system. RMA originates from research in scheduling algorithms for real-time systems by Liu and Layland [Liu73] and has been extended and documented by the Software Engineering Institute at Carnegie Mellon [Klein93]. Systems scheduled with rate-monotonic scheduling (RMS) algorithm could be formally analysed to determine whether meeting real-time deadlines could be guaranteed.

Rapide

The Rapide project from Stanford Univerisity focuses on building large-scale, distributed multi-language systems. It is a concurrent event-based simulation language for defining and simulating the behaviour of system architectures [Luckham95]. The system is based on an executable ADL (EADL), and a toolset is available for supporting the use of this EADL, thus providing for the analysis of system architectures. Rapide adopts an event-based execution model of a distributed, time-sensitive system: "the timed Poset model". Partially Ordered Sets of Events (Posets) provide a formal basis for constructing early life cycle prototyping tools, and later life cycle tools for correctness and performance analysis of distributed systems.

PARSE

The aim of PARSE (PARallel Software Engineering) is the development of techniques and tools to support the production of closely coupled parallel systems [Gorton94][Gorton95]. It is a multi-stage software engineering methodology, and all of the various software engineering stages are based on the PARSE process graph design notations [Gorton96].

The PARSE process graph notation is an object-based design notation, where system components are decomposed into different types of process objects [Gray94][Gorton93]. The interactions between these objects are based on message passing, and the various modes of communication can be specified via the notations available.

This methodology has the following features:

- Systems are composed using a graphical design notation, which enables process structures and their precise interactions to be hierarchically constructed.
- PARSE designs are language and architecture independent.
- PARSE designs can be transformed into formalisms such as Petri nets or CSP to provide for design verification [Gorton96a] [Jelly95].

2.1.1 Methodologies for Dynamic Software Architecture

The primary work of the various software architecture research groups presented so far has revolved around Architecture Description Languages (ADLs). However, most of these approaches are limited to the specification of static systems. These ADLs cannot adequately handle the design of concurrent systems which incorporate the dynamic creation and deletion of processes and of communication paths [Liu98]. Similarly, the dynamic reconfiguration of systems can not easily be captured in the design. In many distributed systems such as retail banking systems, factory automation, and telecommunication, the provision of these facilities is important.

One exception is Darwin [Magee95] which allows the specification of runtime architectural changes. The Imperial College's work on Darwin focuses on the separation of program structure and algorithm behaviour. Darwin is also a configuration language for the Regis [Magee94] distributed programming environment. This group is also currently investigating the use of labeled transition systems for reasoning about the behaviour of Darwin-structured programs [Cheung94]. The Software Architect's Assistant [Ng95] is a CASE tool for the Regis programming environment which does not yet support automated formal analysis and verification of design.

Another research group that examines dynamic systems issues is the University of California, Irvine. This project experiments with dynamic architectures by building a prototype tool that supports the construction and run-time modification of software architectures in an event-based style called C2 [Taylor96] where components communicate via connectors. This approach provides an imperative language for modifying architectures [Oreizy98] and the prototype of the supporting tool ArchStudio provides interactive tools for software architects to describe architecture and architectural modifications. Then, an 'Extension wizard' enacts the runtime modifications. This tool does not provide for the complete analysis and verification of the dynamic software architecture using an explicit formalism, and is only limited to checking invariants from the C2-style rules. This tool also restricts architects to using a specific language: the Java-C2 class framework.

2.1.2 Formal Description Techniques for Software Architectures

Estelle [Diaz89], LOTOS [Leduc94] [Pecheur92] and SDL [Faergemand91] are three formal description techniques (FDTs) that first came out in the late 80's and conform to ISO standards. Both Estelle and LOTOS were developed within ISO for the specification of OSI protocols and services [Ansart89]. However, the typical modeling elements such as modules and interaction points of Estelle and LOTOS means they are typical ADLs and can be used in all software architecture modeling. Estelle supports hierarchical structuring of system components, and has supporting tools which include a syntax directed editor, symbolic debugger and code generators for C and ML. There are also attempts at LOTOS tools including the European/Canadian LOTOS Protocol Tool Set (Eucalyptus) project [Eucalyptus97].

SDL (Specification and Description Language), recommended by CCITT for describing functions of telecommunications systems, is based on experience of describing systems as communicating state machines. Since 1976, SDL has evolved from an informal drawing technique to a formal description technique, and several commercial tools exist [e.g. Faergemand91] which generate code directly from SDL descriptions. SDL has hence received wide use especially in the telecommunication sector [Belina91] [Saracco89].

Looking very similar to a flow chart, SDL has provision for the specification of input and output between different processes. Recently, SDL has also been extended to incorporate object-oriented features [Faergemand94] [Faergemand93] such as process typing, process instance sets, and inheritance. The consideration for dynamic process creation and termination is limited to the description only, no validation process has yet been supported.

Rice and Seidman presented a formal model for Module Interconnection Languages (MILs) [Rice94]. This model formalises the design of hierarchical module structures. The model is specified by a collection of Z schema type definitions that is invariant across all applications. Any particular application then is described by specifying the values of generic parameters and adding application-specific declarations and constraints to the schema definitions. Rice and Seidman have applied their technique to describe the ADLs CONIC [Magee89] and STILE [Stovsky88], where in these ADLs (or MILs), a module interface is described by a collection of named and typed channels. To describe these two ADLs, appropriate values for the general parameters of the formal model is provided, such as those representing notions of interface ports and the types of ports.

2.1.3 Architectural Design Using Object-Oriented Methodologies

In additions to the architectural description languages, distributed software architectures are also commonly specified using various object-oriented methodologies [Low96]. Each of these methods has its own notation (symbols for communicating object-oriented models), process (describing activities to perform in different stages of the development), and tools (the CASE tools that support the notation and the process).

However, there are shortcomings in employing existing object-oriented modeling notations for architectural descriptions of distributed systems [Rasmussen96]. Primarily, existing object-oriented modeling elements are not rich enough to express various architectural features such as synchronisation, inter-module communication types, global control structures and system component reconfiguration [Kramer94]. Further, there has been little formal semantics work carried out in the area of object-oriented modeling. These object-oriented modeling elements are largely informal graphical modeling components, and as a result, there is little support in the formal design analysis and verification area [Harel97].

The Booch Method

Booch [Booch94] defined the notion that a system is analysed as a number of views, where each view is described by a number of model diagrams. The Booch notation is very extensive, and some symbols (such as the cloud for object) are hard to draw. The method also contained a process by which the system was analysed from both a macro and micro development view, and was based on a highly incremental and iterative process.

OMT

The Object Modeling Technique (OMT) is a method developed by James Rumbaugh [Rumbaugh91]. A system is described by a number of models: the object model, the dynamic model, the functional model, and the use-case model, which complement each other to give the complete description of the system. The OMT method also contained a lot of practical description on how to create a system design, taking into account concurrency and mapping to relational database.

OOSE/Objectory

The OOSE and Objectory methods both build on the views of Ivar Jacobson [Jacobson92]. The OOSE method is Jacobson's version of an object-oriented method, and the Objectory method is used for building a number of systems, as diverse as telecommunication systems for Ericsson and financial systems for Wall street companies.

Both methods are based on use cases, which define the initial requirements on the system as seen by an external actor. The use cases are then implemented in all phases of the development, all the way through to system testing, where they are used to verify the system [Jacobson92a]. Objectory has also been adapted for business engineering, where the ideas are used to model and improve business processes.

Fusion

The Fusion method from Hewlett-Packard is a second generation method because it is based on the experiences of many of the initial methods [Coleman94]. Fusion has enhanced a number of important previous ideas, including techniques for the specification of operations and interaction between objects. The method has a large number of model diagrams.

OOA/OOD

The Coad/Yourdon method also known as OOA/OOD [Coad91] [Coad91a] was one of the first methods for object-oriented analysis and design. The method was rather simple and easy to learn, and as such, it worked well to introduce object-oriented concepts to novices.

UML

The Unified Modeling Language (UML) is a joint effort from Booch, Rumbaugh and Jacobson. Aiming to be a standard object-oriented notation and process, it is based primarily on the Booch, OMT and OOSE methods. It also includes concepts from several other methods. For example, the work of Harel on Statecharts [Harel88] has been adopted in the UML state diagrams; parts of the Fusion notation for numbering operations has been included in the collaboration diagrams; the work of Gamma-Helm-Johnson-Vlissides on patterns [Gamma95] and how to document them has inspired details of class diagrams; the concept of Responsibilities came from Wirfs-Brock [Wirfs90]; and Object life cycles from Shlaer-Mellor [Shlaer93].

UML provides model elements from which various diagrams are built from. These different diagrams then present various different views of the system, not unlike the Booch approach. The modeling elements include classes, objects, stereotypes, adornments, and so on. The various diagrams available for modeling are:

- use case diagrams (functionality of system as perceived by external actor);
- class diagrams (software structuring);
- collaboration diagrams and sequence diagrams (class interactions);
- component diagrams,
- deployment diagrams.

The combination of these diagrams thus presents the system in use case, functional, logical, component and deployment views.

Despite the multiple views UML supports, no one single diagram can describe software architecture. To describe software architecture fully, model elements from both class diagrams, deployment diagrams and often more must be employed. For example, many concurrency and synchronization features are not explicitly expressed in any of the UML diagrams.

Various tool supports are available for these object-oriented approaches. Some sample object-oriented CASE tools include Rational Rose [Quatrani98], GDPro [AST98], Software Through Pictures from Aonix [Aonix98], Class Designer from Cayenne Software [Cayenne98] and many more. These CASE tools, depending on complexity (level of sophistication) may support the following:

- visual editor for models
- provide navigation between different views/diagrams
- model syntax checking (consistency checks between different diagrams)
- code generation
- reverse engineering

However, since most of these graphical notations are informal, there has been no adequate support for automated design analysis nor formal model validation.

2.1.4 Comparison

It should be noted that the distinction between methods and tools is an important one. Tools should support methods, and tools should not be built without any real method to support. The accompanying methodology describing the process of how to use design notations and associated tools, is just as important as the notation and tools [Jelly96a].

Architecture definition languages should provide more ways of describing software architecture than just the simple *box and line* drawings. Several important issues such as: the inter-component communication type and protocol, component interfaces, concurrency handling and dynamic communication structures and components are just as important as getting the functional aspects of the components (algorithms and data structures) correct. A well-defined architectural definition language can then be used as a solid basis for developing better tools for designing software architectures and reasoning about the properties of the architecture.

The various object-oriented approaches currently do not address the issues of concurrency handling well. Most current efforts on architectural description languages do not have provisions for dynamic features. Hence there is room for important research advances in these areas.

2.2 Formal Methods for design verification

Many formal methods exist for the specification and verification of concurrent systems.

Specification is the process of describing a system and its desired properties. Formal specification uses a language that has a mathematically defined syntax and semantics. Once specified, the resultant formal model of a system can be subjected to formal analysis, e.g. checked to be internally consistent or used to derive other properties of the specified system.

Two well established approaches to verification are model checking and theorem proving.

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. The check is basically an exhaustive state space search. In contrast to theorem proving, model checking is completely automatic and fast. Model checking can be used to check partial specifications, and so it can provide useful information about a system's correctness even if the system has not been completely specified. The main problem with model checking is the state explosion problem. However, there have been numerous promising approaches that alleviate this problem [Peled96] [Kurshan94] [Elseaidy96].

Theorem proving is a technique where the system and its desired properties are expressed in some mathematical logic. This logic is given by a formal system which defines a set of axioms and a set of inference rules. Theorem proving is then the process of finding a proof of a property from the axioms of the system. Most theorem proving tools are interactive. The B tool [Wordworth96] is an example of interactive theorem prover.

For the design of dynamic systems, it is important to use formal methods to check for the validity of system configuration. For example, in distributed object systems that are typically built on middleware such as CORBA, software components are often dynamically bound and released. It is important to design the software architecture in such a way as to eliminate undesirable properties such as structural deadlock resulting from these dynamic component reconfigurations. Both the model checking and theorem proving approaches are applicable here.

It is not the intention of this survey to include all existing formalisms, but to present those with the capability to describe concurrency and dynamism. These selected formalisms will be presented in the following categories:

- set-theoretic approaches
- logic based approaches
- process algebraic approaches
- state automata approaches.

An assessment of the expressiveness of formalisms for describing dynamic systems with component reconfiguration features will be presented, as well as the availability of the corresponding support tools.

2.2.1 Set-theoretic Approaches

Formal methods such as VDM [Jones89], Z [Spivey92] and B [Wordsworth96] have foundations in set theory. These methods encourage the construction of a model of a system or problem in terms of sets, maps, sequences and predicates.

VDM

The Vienna Description Method (VDM) is one of the more mature formal methods. It has received industry attention: example projects include [Durr95] which employed an object-oriented extension to VDM known as VDM++. The Mural system [Jones91] [Fields92] developed at the University of Manchester supports the construction of VDM specifications and refinements. Users can generate proof obligations to verify internal consistency of specifications. VDM is commonly seen as the predecessor of methods like Z and B.

Z

Z is also based on set theory. Further, to the basic notion of set theory, Z adds the idea that objects in its universe may be categorised into different kinds, and that there is no overlap between distinct kinds. An important device in Z known as the schema allows descriptions of objects to be grouped into units, which can be referred to throughout the specification [Potter91] [Spivey88].

Z is supported by *ZTC*, a PC or Sun based type-checking system available for non-commercial purposes, and *Fuzz*, a commercial type-checker running under Unix and DOS. There are also some more integrated packages that support typesetting and specification integrity checks including Logica Cambridge's *Formaliser*, Imperial Software Technology's *Zola*, which includes a tactical proof system, and York Software Engineering's *Cadiz* (a tool suite for Z that supports refinement to Ada code). ICL's *ProofPower* uses Higher Order Logic [FME98] to support specification and verification in Z [Bowen95]. The wide choice of Z support tools reflects the popularity of the Z formalism.

B

Devised by Abrial, like Z and VDM, B is a model-oriented method based on set theory and refinement theory [Lano96]. B has been designed to cover all the development phases of the software life-cycle, from specification to implementation, with emphasis on modularity and data encapsulation. The formal specification, as well as the design and implementation, are expressed in an abstract machine notation.

The B tool from B-Core supports the B method [B-Core96]. The tool supports the construction of specifications called machines. The method advocates specification reuse, and the tool supports this by providing a library of base machines from which other machine specifications can be extended. Specifications can also be checked for consistency between subsequent refinements via proof obligations and generated into skeleton C code.

The French company Matra Transport is using the B method to design safety-critical software for the driverless trains on the new Meteor line in the Paris Metro [Nairn96] [Dehbonei95].

Discussion

These methods and associated tools model data structures and algorithms well, and have received usage in various applications such as the specification of the IBM Customer Information Control System (CICS) using Z [Hayes91] and using B [Hoare-JP95]. (Although only about one-tenth of the entire system was actually subjected to formal techniques [Bowen95]).

However, these set-theoretic approaches cannot easily verify concurrent systems. In dealing with concurrent and distributed systems, care must be taken to define the notion of correctness. Traditional (deterministic) sequential programs may be viewed as partial functions from inputs to outputs, specification may be given as a pair consisting of a precondition describing the allowed input and a postcondition describing the desired output for these inputs (e.g. models described in Z and B). However, for reactive and non-deterministic concurrent systems, this approach is too limited.

There have been attempts at specifying distributed systems using the B method, however, the resultant models are often awkward [Butler96].

2.2.2 Logic Based Approaches

Temporal Logic

This category emphasizes the use of logical formulas to encode properties of interest in a concurrent system. In the seminal paper [Lamport77], Lamport argues that the requirements that designers wish to impose on reactive systems fall into two categories. *Safety* properties state that "something bad never happens"; *Liveness* properties, on the other hand state that "something good eventually happens". Much work has been produced in developing logics that formalise these informal yet useful notions. The most widely studied is Temporal Logic [Pnueli77] which supports the formulation of properties of system behaviour over time.

There are different types of temporal semantics: interval, point, linear, branching and partial order. Correspondingly, there are variants of Temporal Logic that uses different semantics. The variants include: the TTM/RTTL framework - explicit clock linear logics [Ostroff85], Metric Temporal Logic (MTL) – hidden clock linear logics [Koymans90], and XCTL - discrete time propositional explicit clock logic [Harel-E90]. The various temporal logics can be used to reason about qualitative temporal properties. Safety properties that can be specified include mutual exclusion and absence of deadlock. Liveness properties include termination and responsiveness. Fairness properties include scheduling a given process infinitely often, or requiring that a continuously enabled transition ultimately fires.

Real-Time Logic

Real-Time Logic (RTL) is another logic based formal language for reasoning about events and their times of occurrence [Mok91]. In [Jahanian88], a visual formalism called Modecharts is introduced. Modecharts specify a decidable fragment of RTL, in a state-based fashion. A method is provided for translating Modecharts into computational graphs, from which the verification can be performed. RTL's event occurrence function allows for a rich expression of periodic and non-periodic real-time properties. However, unrestricted RTL is undecidable. It does not treat infinite state systems, nor dynamically reconfigurable systems. RTL formulas impose a partial order on computational actions which is useful for representing high level timing requirements.

Assertional Logic

There are various other assertional logic approaches such as the Real-Time Hoare Logic [Hooman89] which is based on the classical Hoare triples: {q}P{r}, where P is a program, q and r are first order predicates [Hoare69]. Hoare triples can only express partial correctness (properties that hold if the program terminates). This is hence not suitable for distributed systems which must deal with non-terminating programs, and interactions with the environment.

2.2.3 Process Algebraic Approaches

The set theoretic models and logic models discussed in previous sections encourage the construction of a model of a system in terms of mathematical data structures and of the static and dynamic constraints on them. By contrast, the process algebraic approaches exemplified by CSP [Hoare85] and CCS [Milner89] allow a system to be modeled by a collection of processes which communicate with one another.

CSP

Communicating Sequential Processes (CSP) devised by C.A.R. Hoare presents a process as a mathematical abstraction of the interactions between a system and its environment. Events, communication between processes, and non-determinism inherent in concurrent systems can be modeled in CSP. Further, in addition to the usual logic, functions and set operators, there are various other operators for describing processes and their interactions such as: sets of messages, event ordering, choice, parallelism, interleaving, chained to, subordinate to, interrupted by, restartable, repeat, satisfies, assignable and accessible. The primary reasoning mechanism of CSP is the concept of execution traces, where the sets of all sequences of events in which a process can participate can be analysed, and checked for certain properties such as deadlock.

FDR [Roscoe97] from Formal Systems Europe is a model and refinement checker for CSP.

CCS

CCS is also aimed at describing concurrent processes. The syntax of CCS is simpler than CSP, in that Milner has kept the number of operators in CCS to a minimum, and believes that higher level constructs such as conditionals and data structures can be encoded using primitive operations.

Similar to CSP, CCS captures the ordering of events and interleaving actions of concurrent processes. The creation and termination of processes can also be described. However, the communication links between processes is pre-determined, and dynamic reconfiguration of communication can not be captured here.

The Concurrency Workbench [Stevens98] is a CCS model checker.

π-Calculus

Milner devised the π-Calculus [Milner91] [Milner92] [Milner92a] after his work on CCS, and π-Calculus is often seen as a descendent of CCS. The basic notion of the π-Calculus is the idea of 'naming', where 'names' can be freely passed around. This powerful notion enables the modeling of dynamic reconfiguration of system components and communication links.

Like CCS, there has also been extensive research work into equivalences of models in the π-Calculus work, which provides the basis for model refinement [Carrington94] [Morgan90].

The Mobility Workbench is a model checker for the π-Calculus [Victor94]. It allows syntactical checks on π-Calculus expressions, as well as equivalence checks on two π-Calculus expressions at a time, in the manner defined by the equivalence theories [Sangiorgi96]. Further, it provides facilities for deadlock checks.

Ambient Calculus

Devised by Luca Cardelli in 1997, Ambient Calculus [Cardelli98] is an extension to π-Calculus which aims to capture the relationship between mobile processes and run-time environment. Novel concepts here include the 'in' and 'out' operators which describe the actions of a process entering into and exiting from a computing processor. Ambient Calculus seems promising in describing mobile processes, however, the model is not yet mature, and there are no support tools at present.

2.2.4 State Automata Based Approaches

State machines have been a useful modeling technique in many branches of engineering. More powerful variants such as Statecharts by David Harel [Harel88] and Petri Nets [Peterson81] have provision for describing concurrency.

Statecharts

A state diagram is a bipartite graph of states and transitions. It shows the sequence of states that an object or an interaction goes through during its life in response to received stimuli, together with its responses and action.

Harel's work on Statecharts [Harel84] was a substantial improvement on the traditional flat state machines. It also contains features such as hierarchical decomposition and nested states, as well as the specification of concurrently executing components using substates. The interactions between these components can also be specified using message passing.

Statemate is a widely used tool for Statecharts [Harel90].

Recently, the Statechart formalism has been incorporated as a part of the dynamic behaviour specification notation in the Unified Modeling Language (UML) [Penker98] [Harel97].

Petri nets

Most theoretical work on Petri nets is based on formal definition of Petri net structures in terms of bag theory. However, the graphical representation of the Petri net structure is much more powerful and useful for modeling systems.

A Petri net is a representation of a Petri net structure as a bipartite directed multigraph, consisting of places, transitions and directed arcs [Petri62].

The basic Petri net consists of places and transitions. The firing of transitions depends on the availability of tokens. Petri nets can be easily used to model concurrent processes, whereby the tokens represent active threads of control, places represent the different states thread executions may be in, and various synchronization mechanisms may be modeled via the use of transitions.

Figure 2.1 illustrates the well known reader/writer problem [Petersen81], where writer processes must mutually exclude all other reader and writer processes, but multiple reader processes can access the shared data simultaneously. This solution allows n readers to read at a time.

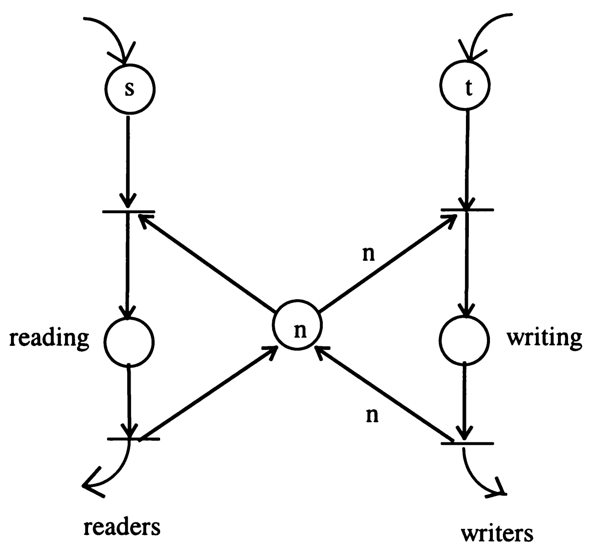


Figure 2.1. Reader/Writer Problem with Bounded Number of Readers Initially s Readers and t Writers

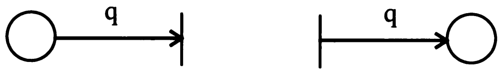
However, if an unbounded number of readers and writers are assumed, and that we want to allow an unbounded number of readers at a time, then the system cannot be represented by Petri-nets. In fact, it is necessary for readers to keep count of the number of readers reading, and increment or decrement this counter when it starts or finishes reading. This can be modeled by a place with a number of tokens equal to the number of readers. This means that for a write to begin, it must be able to read for an empty place. However, there is no mechanism in Petri nets which allows an unbounded place to be tested for zero. Thus, if there is an unbounded number of processes entering the system, then it cannot be represented by Petri nets.

Inhibitor arcs

To overcome this problem, the additions of inhibitor arcs allow zero testing. It has a small circle rather than an arrowhead at the transition, where the small circle means 'not'. The firing rule is a transition is enabled when tokens are in all of its (normal) inputs and zero tokens are in all of its inhibitor inputs. The transition fires by removing tokens from all of its (normal) inputs.

Self-modifying Net

The self-modifying Petri net is another extension to the Petri net, which permits the labelling of an arc with the name of a place, to denote that the enabling of the arc depends on the number of tokens present in that particular place. If the place has no tokens, then the arc does not exist. Formally, a self-modifying net is defined like an ordinary Petri net as a bipartite multi-graph having edges of the form:



If q = 1 then the firing rule of the transition is defined as in the ordinary case. But q is also allowed to be the name of an arbitrary place of the net. In this case, the number of tokens to be moved from or to the place equals the actual number of tokens in q. Therefore, self-modifying nets are able to modify their own firing rules [Valk77]. Figure 2.2 illustrates how a self-modifying arc can be disabled.

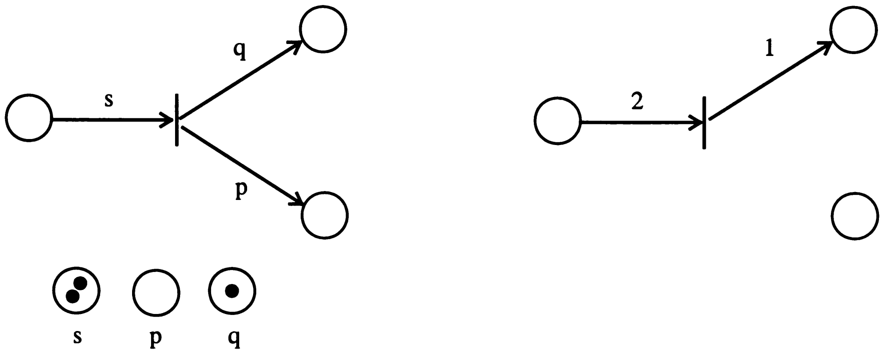


Figure 2.2. Self-Modifying Arc

We now can provide an alternative solution to the readers/writers problem.

- the *in* transition controls the entry of reader and or writer processes, so that the *n* place can keep count of the actual number of processes in the system.
- the *w'* place is the complementary place of *w*. The presence of a token in place *w'* means the absence of writer processes. In this case, arc *x* is enabled, and the *in* transition can add tokens to place *s*, which can be used to enable transition *b* (reader) or accumulated to enable transition *y* (writer).

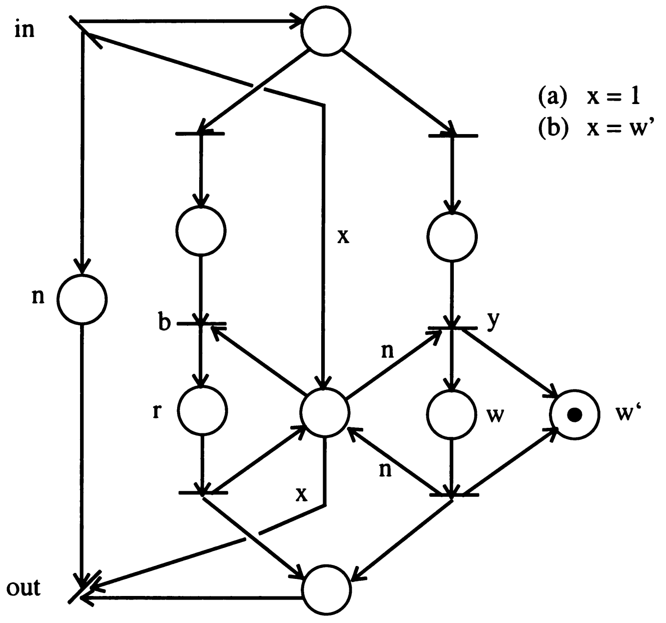


Figure 2.3. Reader/Write Problem Revisited - Self Modifying Net Solution

Other Petri Net Variants

There are many other variants of the basic Petri Net beside the inhibitor arcs and Self-Modifying Nets. These include:

- Coloured-Petri Nets [Jensen92], these are high level Petri Nets often used to model data types,
- Stochastic Petri Nets [Ciardo94], for timing sensitive systems, and
- Object-oriented Petri Nets [Lakos91].

Each of these variants have varying degrees of expressiveness, and thus have an impact on the complexity of corresponding Petri Net tools.

2.2.5 Discussion

There is a general consensus among the software industry that formal methods are difficult to use. These are often attributed to the complexity in formalism, and the lack of support tools [Holloway96].

Hence, there is a need for good supporting tools for various functions such as:

- (visual) modeling
- model checking (e.g. safety, live-lock, deadlock, reach-ability)
- design analysis
- model refinement
- code generation.

2.3 Distributed Computing Environments and Language Support

Existing operating systems do not have adequate support for distributed software engineering [Liu96]. Network operating systems such as Windows NT [Custer95] and Novell enable the sharing of network resources, however, not at a transparent level. Distributed operating systems such Mach [Rashid89] and Amoeba [Renesse89] are not mature enough to be used outside the research community. One exception is Chorus [Rozier90] where the CHORUS/COOL ORB has been recently acquired by Sun Microsystems and is receiving wide usage [Lea93].

Instead of adding more functionality to existing operating systems, support for distributed software engineering has been introduced at the middleware level. See Figure 2.4. There is also support at a programming language and/or environment level, e.g. Java.

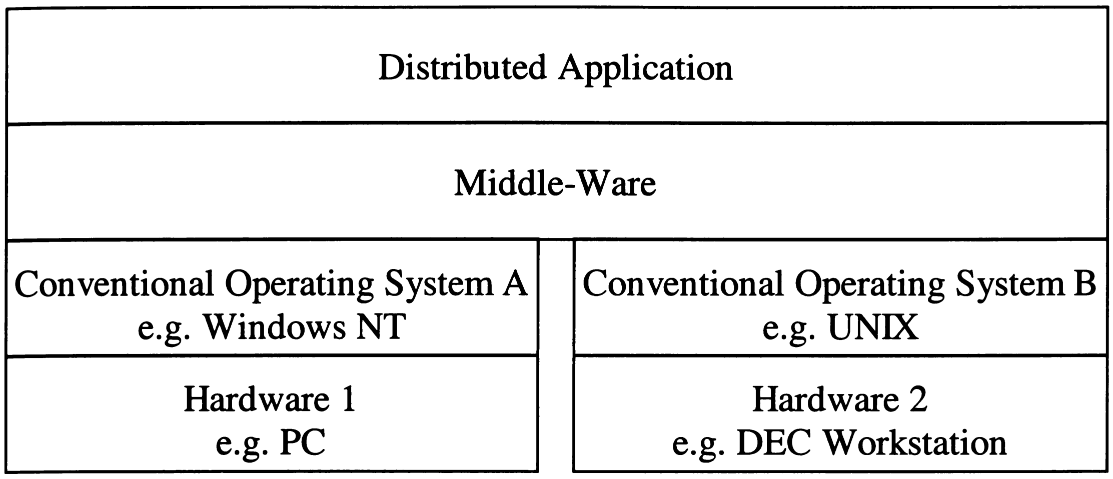

Distributed Application
Middle-Ware
Conventional Operating System A e.g. Windows NT
Conventional Operating System B e.g. UNIX
Hardware 1 e.g. PC
Hardware 2 e.g. DEC Workstation

Figure 2.4. Layers of Support for Distributed Software

2.3.1 Middleware

In order to provide support for distributed systems, various middleware layers have been developed. Middleware provides the basic infrastructure for abstracting the communication layer, thus simplifying the software engineering process. Existing middleware includes CORBA, DCOM from Microsoft, and the longer standing DCE.

CORBA

CORBA is a de-facto industry standard for distributed object systems development [OMG95]. The Object Management Group (OMG) brought vendors and end users together to agree on the technical content of this distributed object architecture. The CORBA specification details how objects should be able to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in heterogeneous and homogeneous environments.

The ORB (Object Request Broker) is central to the CORBA architecture. An ORB is a software component whose purpose is to facilitate communication between objects. It does so by providing several capabilities. These include locating a remote object when given an object reference, and the marshaling of parameters and return values to and from remote method invocations. The ORB thus provides platform-independence to distributed CORBA objects.

Language independence is achieved in CORBA via a common object Interface Definition Language (IDL) which defines the types of objects according to the operations that may be performed on them and the parameters to those operations that is part of CORBA. The following is a sample OMG IDL of an *Account* interface:

```
// Account.idl

// Forward declaration of Account interface.
Interface Account;

#ifndef Account_idl
#define Account_idl

// sequence of Accounts
typedef sequence<Account> AccountList;

# include "Customer.idl"

interface Account {

       // This Account's account number.
       readonly attribute string accountNumber;

       // This Account's current balance.
       readonly attribute float balance;

       // return list of Customers who hold this Account
       CustomerList getCustomers();

       // Withdraw the given amount from this Account.
       // Returns new Account balance.
       float withdraw(in float amount);

       // Deposit the given amount into this Account.
       // Returns the new Account balance.
       float deposit(in float amount);
};
#endif
```

Interfaces described in IDL can be translated into any programming language. CORBA applications and components are thus independent of the languages used to implement them. For example, a client written in Java can communicate with a server implemented using C++, which can in turn communicate with another server written in COBOL.

Existing implementations of CORBA include Iona Technologies's Orbix [Iona95], BBN's freely available Corbus [BBN98] and Hewlett-Packard's ORB Plus [HP96].

DCE

The Open Software Foundation's Distributed Computing Environment (DCE) is another middleware product that supports distributed computing. DCE enables computers from different manufacturers operating on different system software platforms to interact, sharing data and applications transparently across networked, distributed environments.

The fundamental communication mechanism of DCE is the Remote Procedure Call (RPC). RPC allows direct calls to procedures on remote systems as if they were local procedure calls. This simplifies development of distributed applications by eliminating the need to explicitly program the network communications between the client and server. The DCE RPC mechanism masks differences in data representations on different hardware platforms, allowing distributed programs to work transparently across heterogeneous systems. The primary difference between DCE and CORBA is that DCE is not necessarily object-oriented. There are many implementations of DCE around, most notably the ones from Transarc [Transarc98] and Siemens [Siemens97].

DCOM

The Distributed Component Object Model (DCOM) [Microsoft98] is a protocol that enables software components to communicate directly over a network. Previously called "Network OLE", DCOM is designed for use across multiple network transports, including Internet protocols such as HTTP. DCOM is based on the Open Software Foundation's DCE-RPC specification and will work with both Java applets [Horstmann98] and ActiveX [Chappell96] components through its use of the Component Object Model (COM). For example, a developer could use Java to build a Web browser applet that calculates the value of a portfolio of securities, using DCOM to communicate stock values to the applet in real time over the Internet.

ActiveX [Chappell96] controls are among the many types of components that use COM technologies to provide interoperability with other types of COM components and services. ActiveX controls are the third version of OLE controls (OCX), providing a number of enhancements specifically designed to facilitate distribution of components over high-latency networks and to provide integration of controls into Web browsers. These enhancements include features such as incremental rendering and code signing, to allow users to identify the authors of controls before allowing them to execute.

2.3.2 Distributed languages

Java

Software system components written in Java can be transported across the World Wide Web via HTTP, and executed on different types of machines. Java's portability is due to the concept of a Java Virtual Machine which acts as a translator that transforms general Java platform instructions into platform dependent executable code. Java thus supports the construction of distributed systems, although the actual transporting of Java software components is by no means a transparent process.

In addition to Java's portability, Java Remote Method Invocation (RMI) is a CORBA-like architecture that enables distributed software construction. One advantage of RMI is that it supports the passing of objects by value, which is a feature not currently supported by CORBA. A disadvantage is that RMI is a Java only solution, i.e. both RMI severs and clients must be written in Java. For all Java applications, particularly those that benefit from the capability to pass objects by value, RMI is a good choice. For those where interoperability will be a concern, CORBA is the obvious choice.

2.4 Conclusion

This chapter has reviewed existing work on support for the engineering of distributed systems. The three groups of work are:

- software architecture methodologies
- formal methods for architectural design verification
- distributed computing environments

We have seen that there is an abundance of theories and models for describing and analysing concurrency. However, it is not clear how these theories can best be applied in real world systems. There are also robust distributed computing environments present, however, there is no adequate methodology guiding the design of distributed systems that runs on these environments [Gorton97a].

There is clearly a need for a software architecture design and analysis methodology for dynamic distributed systems. This methodology should be supported by an integrated toolset, for the purposes of prototyping architectural design, and automating the formal analysis of these designs.

3. Dynamic PARSE Design Methodology

The PARSE software engineering methodology has been introduced in section 2.1. This chapter presents Dynamic PARSE (PARSE-D) which is an extension of PARSE that aims to cater for not only the design of static, closely-coupled parallel software (as supported by existing PARSE), but also the design of loosely-coupled distributed systems. These loosely coupled distributed systems often exhibit dynamic reconfiguration features such as those commonly found in mobile communication systems and various types of software applications running across a network. These dynamic features add to the complexity of the system greatly, and have severe implications for the system's correctness at run-time. It is thus important for software engineers to capture these dynamic reconfiguration features at an early stage of the software development process, and thus refining and implementing the system with these distribution and concurrency issues in mind. The representation of these dynamic features can also be used for design analysis and verification.

A presentation of PARSE [Gorton95] in sufficient detail will firstly be given. This is followed by a presentation of Dynamic PARSE design notation and the associated usage rules [Liu96]. The Dynamic PARSE design methodology and a sample design will also be presented.

3.1 The PARSE Methodology and Design Notation

PARSE (PARallel Software Engineering) is a software engineering methodology to facilitate the design of reusable parallel systems. This methodology exhibits the following features:

Graphical Design Notation: Systems are composed using a graphical design notation, which enables process structures and their precise interactions to be hierarchically constructed.

Language and Architecture Independence: PARSE designs do not rely on any specific programming language features or target machine architecture.

Formal Verification: PARSE designs can be mechanically transformed into Petri nets to provide the potential for design verification.

Code Generation: PARSE designs can be mechanically transformed into skeleton program code: an example is Occam [Gorton96b].

A PARSE process graph thus acts as a basis for subsequent software engineering stages such as design verification and code generation.

PARSE process graphs depict the process partitioning and communications relationships between processes, together with the role of each process in the system. Conceptually a process graph comprises several concurrently executing processes that interact and synchronise via message passing. Figure 3.1 shows the designs notations of PARSE process graphs.

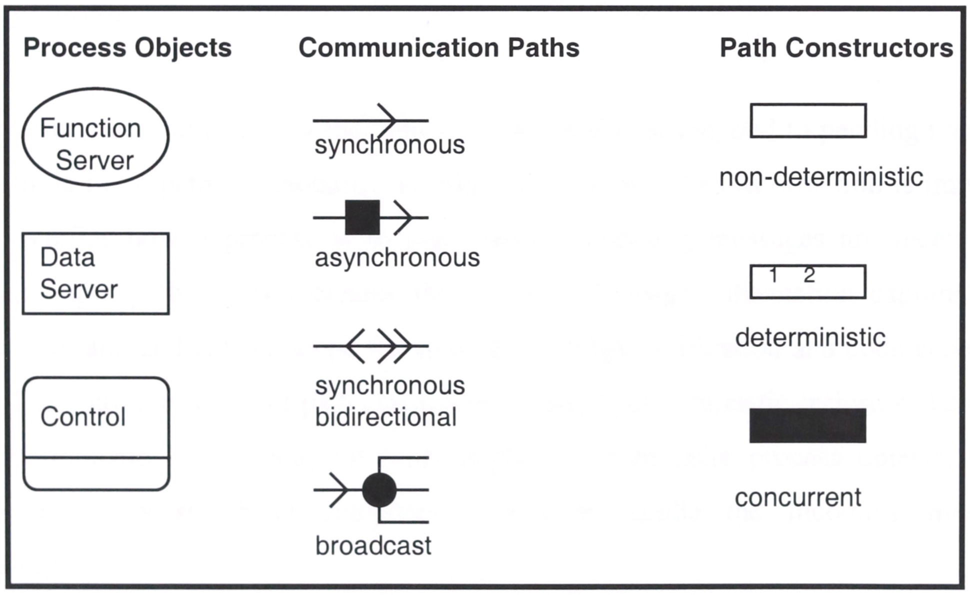


Figure 3.1. Summary of the PARSE Process Graph Notation

A system consists of a set of process objects, where these process objects have some encapsulated data and or functionality (object) as well a thread of control (process).

There are three fundamental categories of process objects, namely Function server, Data server and Control process objects. Essentially, Function servers take a passive role in the system behaviour and basically encapsulate some well-defined functionality required by the system; Data servers also take a passive role and encapsulate some required data and associated access methods. Control process objects have an active role in the system and implement tasks such as the distribution of work and synchronisation of the system. The process notation thus represents a simple formalism which captures the design heuristics used by software engineers when partitioning a problem into processes. Further, it encourages designers to create the passive process objects Function and Data servers which may be candidates for reuse in subsequent developments.

The PARSE process graph also supports four types of communication paths between processes. These are as shown in Figure 3.1 and include synchronous, asynchronous, bi-directional synchronous, and broadcast. In addition to process objects, some communication paths may have external objects as the source or destination. Typical external objects include hardware devices or other software subsystems outside the scope of current development. These external objects are represented by a named, solid vertical bar, and it is only the interface of these external objects that is important. Hence, it is only necessary to specify the type of communication path connecting process objects to external objects.

Path constructors indicate how individual processes should respond to pending messages on multiple input paths. Annotating a design diagram with this information is important as it specifies how a process is to react when competing messages are received on different input paths. This increases the amount of design information captured in a design diagram, and creates scope for automated design verification and code generation tools. There are four types of path constructors, namely deterministic (prioritised choice), non-deterministic, and concurrent (only applied to composite process objects, where independent process object components exist to handle the incoming messages separately).

Some of the other features of PARSE process graph include:

- The specification of replicated instances of a process object by using an index after the process name.

- Ability to specify regular process structures such as pipelines and matrices.
- Hierarchical decomposition of process objects to manage complexity in design diagrams.

3.2 Dynamic PARSE Design Notation

The Dynamic PARSE (PARSE-D) process graph notation (Figure 3.2) is designed to supplement the static PARSE process graph notation discussed in section 3.1, while maintaining the machine and language independence property. Some of the usage rules present in PARSE also apply to Dynamic PARSE. The resulting extended notation enables the design of distributed and parallel systems incorporating dynamic features. In distributed object systems commonly built on middleware such as CORBA, software components are often dynamically bound and released. It is important to explicitly specify this feature.

Dynamically Created Process Objects: function server, data server, and control processes all may be created and deleted dynamically. The existing behavioural and functional rules applying to their static counterparts also apply to them. In addition to the *inherited* functionality of their static counterparts, these dynamic process objects may enter and or exit the system at run time. This should not affect the execution of other processes. Also, communication paths going into and/or coming out from dynamic process objects are also dynamic in nature. These communication paths are set up when the associated process objects are created, and are destroyed when process objects terminate.

Dynamic process objects are often replicated. Each replicated instance has the same process-internals. Different instances may be created and terminated at different times throughout the run-time. The series of numbers enclosed by square brackets [0..n] denotes the range of the number of instances of the object that may be present in the system at any one time. The symbol *n* may be replaced by a constant integer, or by default, is the maximum number of thread instances a process may have as defined by the underlying system. When expanded, each instance has an associated subscript. This provides a way to uniquely identify each instance of the process.

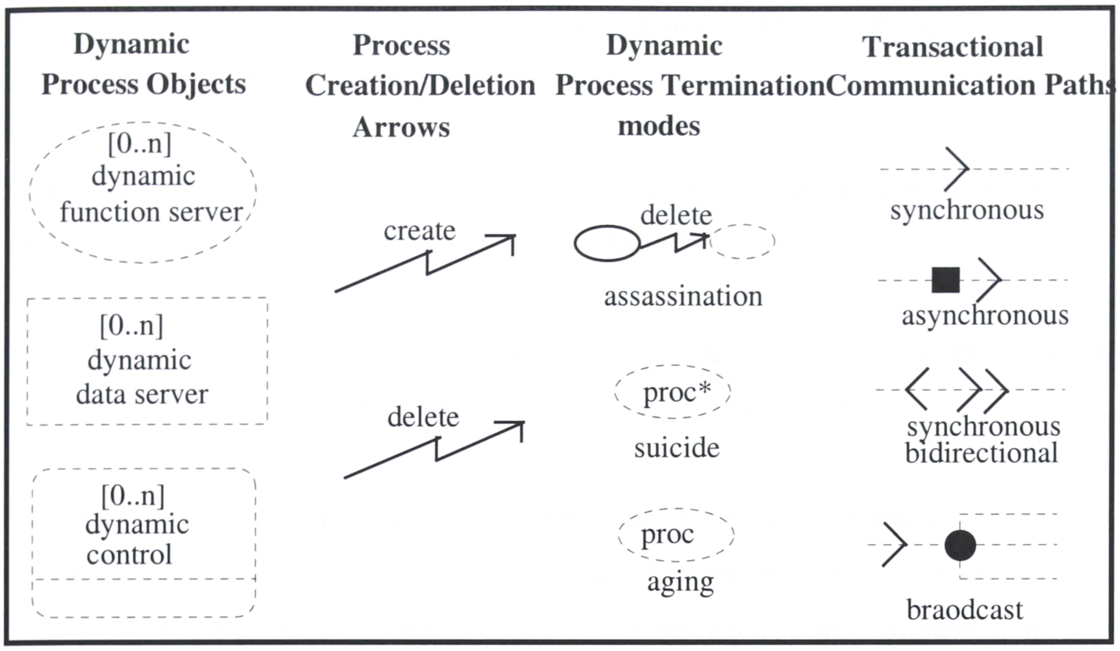

Dynamic Process Objects
[0..n] dynamic function server
[0..n] dynamic data server
[0..n] dynamic control
Process Creation/Deletion Arrows
create
delete
Dynamic Process Termination modes
delete
assassination
proc*
suicide
proc
aging
Transactional Communication Paths
synchronous
asynchronous
synchronous bidirectional
braodcast

Figure 3.2. Summary of Extended-PARSE Process Graph Notation

Creation/Deletion of Dynamic Process Objects: Function servers and control process objects may create and delete dynamic process objects by invoking *create* and or *delete* signals. They are shown via the *twisted arrow* notation. There are two rules of usage:

- the process object at the *invoked* end of a creation/deletion arrow must be of *dynamic* type.
- the process object at the *invoking* end of a creation/deletion arrow must be an *active* process object. This means the passive data server and function server is excluded.

Termination Modes of Dynamic Process Objects: There are three ways that dynamic process objects may exit from the system: *assassination*, *suicide*, and *aging*.

Assassination: Any running process can kill a dynamic process. This ensures that Dynamic PARSE designs are machine and language independent. In different distributed operating systems, there are limitations imposed on deletion of processes by other processes. For examples, in some operating systems, only parent processes may kill their own children processes. In others, and especially when the processes are running in different memory locations such as a distributed object system across a network, as long as the process id is known, any process may kill it.

Suicide: A process or a thread may terminate its own execution. For example, threads in Windows NT may terminate themselves by calling the Win32 function: *ExitThread*.

Other heavy weight processes or a subsystem component may exit the system by itself due to certain system exceptions.

Aging: This is the default termination mode. The created process dies from aging when it completes its work. This occurs naturally, hence the term *aging*.

The notation provides a rich set of notations for describing all possible ways of termination, and leaves the design decision for the software engineer.

Dynamically Created Communication Path: The four types of communication paths can be used to connect to dynamic process objects. When both the sender and receiver process objects are static process objects, the semantics of the communication path remains the same as in PARSE, and any existing PARSE design rules apply here. However, when either the sender or receiver process object is dynamic, the life-time of the communication path is dependent on the associated dynamic process object. For example, in Figure 3.3, the communication path *data* is not valid at all times. It is valid when both the sending process and the receiving process are present in the system. Hence, in this case, the *data* path is set up only after the creation of *sender* by *control* and the creation of *receiver* by an external entity.

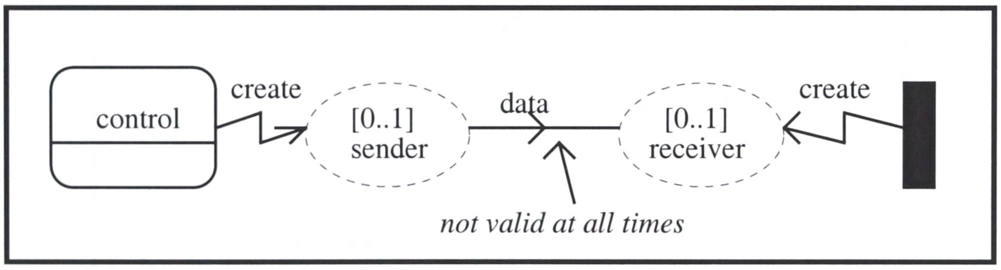


Figure 3.3. Implicit Dynamic Behaviour of Communication Paths

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

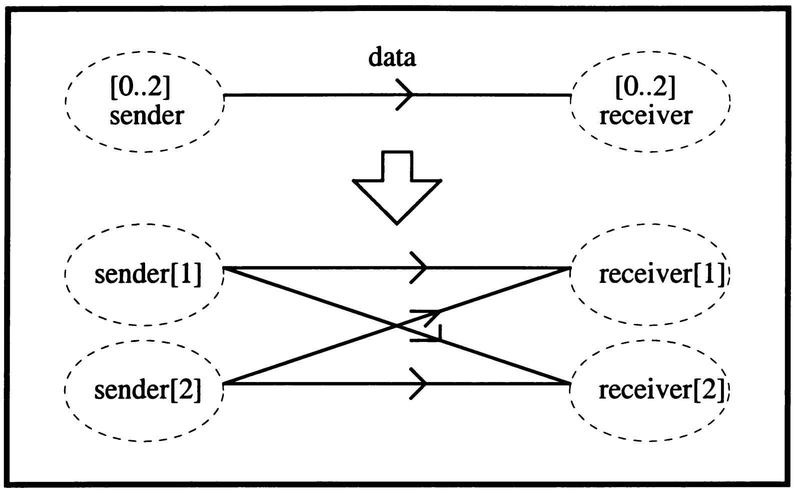


Figure 3.4. Replication of Dynamic Communication Path

The existing replication rules and notations in PARSE also apply to dynamic processes in Dynamic PARSE. That is, by default, all associated processes are fully connected by communication paths when all the dynamic process object instances exist. Path restriction can also be used here to overwrite default behaviours (see Figure 3.5).

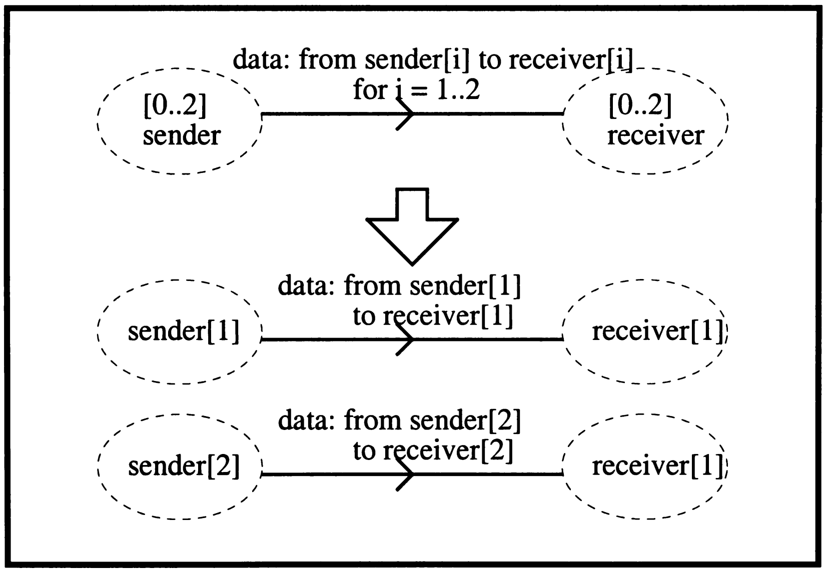


Figure 3.5. Path Restriction in Dynamic PARSE

Transactional communication path: Software designers can explicitly show that the communication between two processes is of a transactional type by using dotted arcs (see Figure 3.6). These are different to the ordinary communication paths in the sense that they are set up only when they are needed to transfer messages. As soon as the transfer is complete, the path is no longer valid. Hence, the life span of a communication path is not dependent on the life spans of the processes using it, but is dictated by the activity of transferring the message. In a database system, a typical example would be a *transactional update* to a data store.

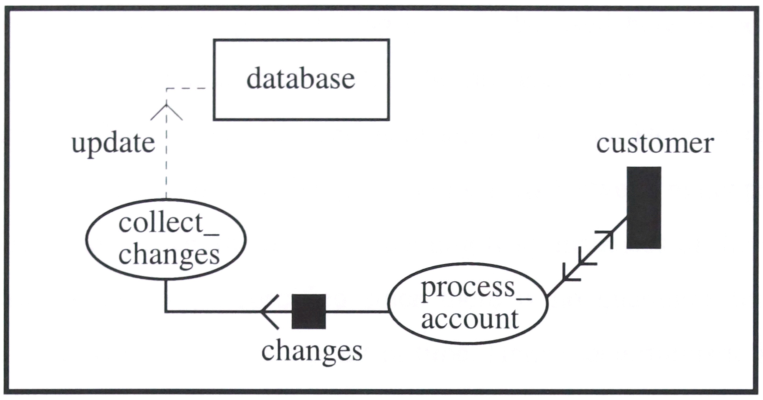


Figure 3.6. Transactional Communication Path Example

The period of validity of the communication path depends on the associated process internals, where process internals can be explicitly defined using an internal behaviour specification language such as the PARSE Behavioural Specification Language (BSL) [Gray94]:

```
PROC database
     SEQ
          -- other processing
          sy-receive(update)
          -- other processing
     ENDSEQ
ENDPROC

PROC collect_changes
     SEQ
          -- other processing
          sy-send(update)
          -- other processing
     ENDSEQ
ENDPROC
```

In this case, the transactional communication path *update* is not valid while *other processing* are happening.

Communication Path Constructors: Not all four types of path constructors may be applied when dynamic process objects are associated. Consider the example of an incorrect use of path constructor in Figure 3.7: the *updates* communication paths may be replicated depending on the number of worker processes present in the system. Hence, there may be multiple paths going into the data store, which need to be handled non-deterministically. The data server receives update messages from workers randomly.

Concurrent and deterministic input handling can not be used here as concurrent input handling implies there is concurrency within the process. That is, the process may be further decomposed. This cannot be used here since the data store is primitive. With deterministic input handling, the receiving process selects between ready paths according to path labels. However, when the paths are coming from instances of the same dynamic process, the priority cannot be specified since there is no guarantee that a particular instance of the process is present at a point in time. Hence, deterministic input handling cannot be used here.

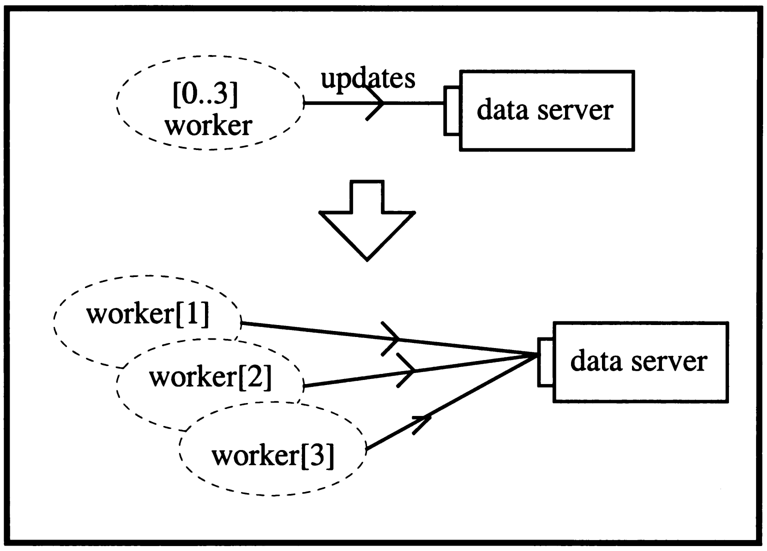


Figure 3.7. Dynamic Updates to Data Server Requiring Non-deterministic Constructor

Hence, path constructors are not to be used in conjunction with dynamic process objects. All multiple inputs to dynamic process objects are handled non-deterministically.

Note: figure 3.7 illustrates an incorrect use of path constructor. Further, the figure is not syntactically correct.

3.3 The Dynamic PARSE Design Methodology

The Dynamic PARSE software design methodology provides a hierarchical, object-based approach to the design of dynamic software architecture. A system is decomposed into a set of concurrently executing *process objects* which communicate via message passing. These process objects can also be hierarchically decomposed in order to handle complexity in design. The graphical design notation is used to capture the architectural (structural and dynamic) properties of the required distributed software system.

A typical design process involves the following steps:

1. identify the various system components in the system.
2. classify the system components to be one of: active control process object (either composite or primitive), passive function process object, and passive data server.
3. identify any process objects that are created and or deleted dynamically. Denote the dynamic process objects and the corresponding creation and deletion actions by using Dynamic PARSE design notations.
4. specify the interaction between the process objects using the 4 different types of communication paths: synchronous, asynchronous, bi-directional (request-reply), and broadcast.
5. use dynamic communication paths to specify any re-configurable communications.
6. For each composite process object, identify its component process objects and the interaction modes. This step is repeated until all composite process objects at all levels have been specified in terms of primitive process objects.
7. for each static primitive process object, indicate the order of handling of all incoming messages to that process object using path constructors.

3.3.1 Sample Design - A Lift Controller

This section illustrates a typical Dynamic PARSE design. The example chosen here is a simplified version of the *Ubiquitous Elevator Problem* discussed in [Howland-Rose94], where the software structure changes according to client requests. It will be shown how this dynamic change to software structure can be easily described using the Dynamic PARSE design notation.

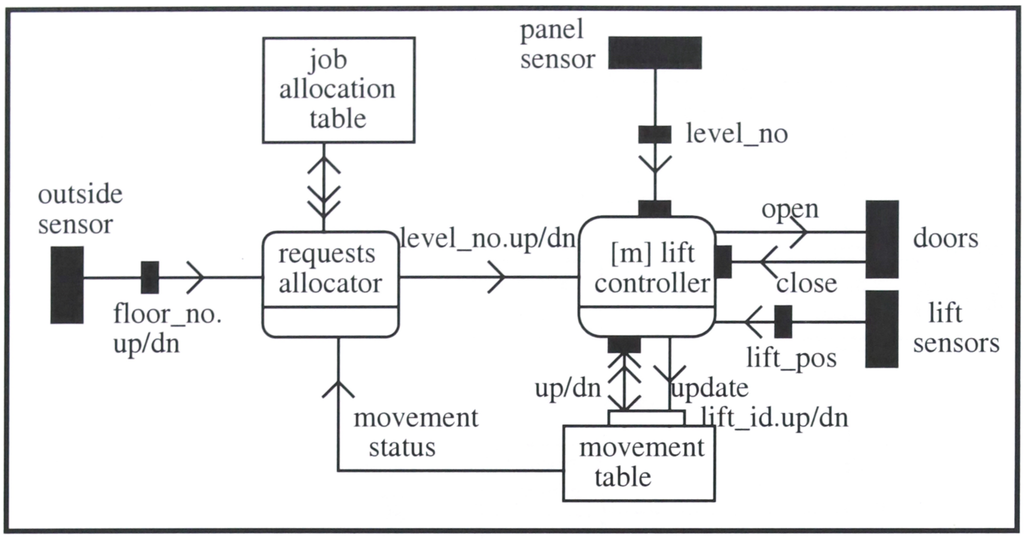


Figure 3.10. Top Level Process Graph of the 'Lift Problem'

The requirement is to design software which controls passenger elevators. Elevators can be called via *up* or *down* buttons on each floor. Inside an elevator, passengers may select destination floors by pressing buttons on a panel. However, while the elevator is in motion, only requests to go to floors in the same direction are accepted by the controller. Figure 3.10 shows the top level design of the system, after carrying out the design steps 1 - 5 listed in section 3.3.

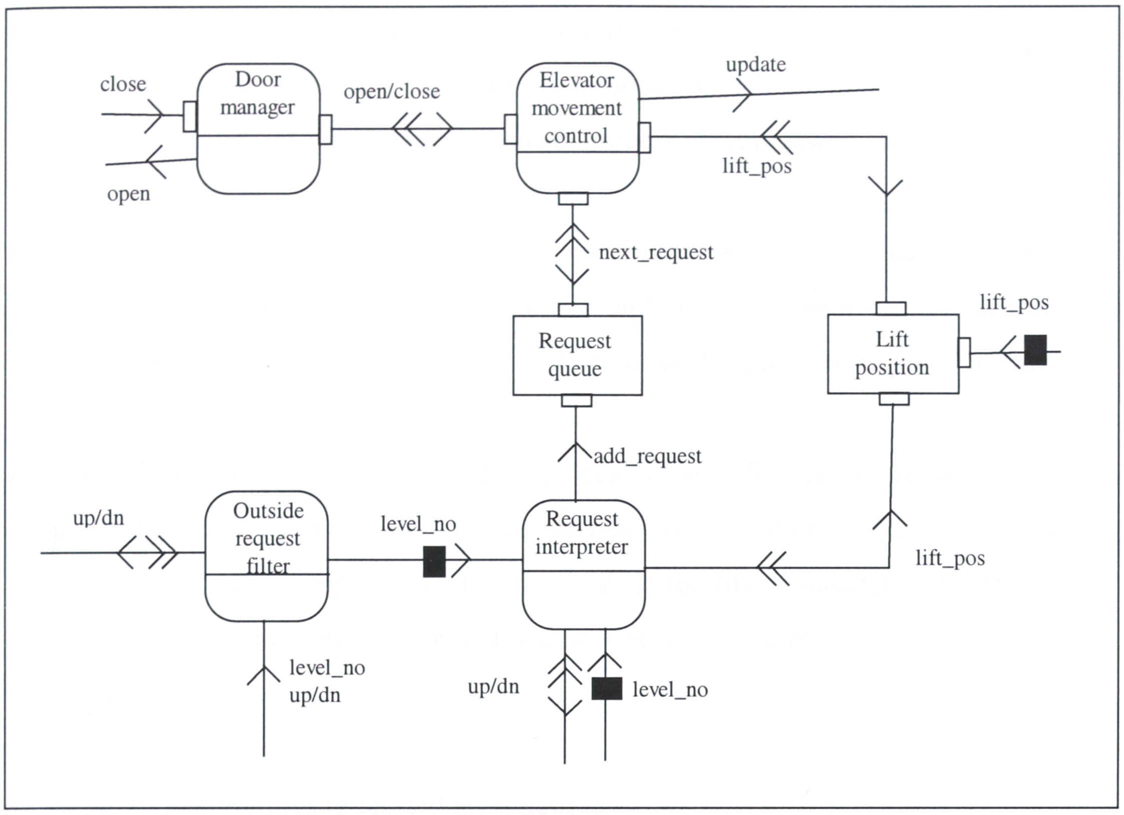


Figure 3.11. Decomposition of the 'Lift Controller' Process

Figure 3.11 is derived by decomposing the Lift Controller process, i.e. carrying out step 6 presented in section 3.3.

For a set of *m* lifts, there are *m* lift controller processes, each controls their own lifts (Figure 3.10). There is also a process *requests_allocator*, which delegates tasks to different lifts upon the receipt of requests from outside sensors. Notice there is only one path being used at any time between *requests_allocator* and *lift_controller*. There is always only one lift allocated the job to service a request. The *request_allocator* chooses a lift by reading the *movement status* of all lifts, as well as the *job_allocation_table*. Thus ensuring all lifts have approximately equal number of requests to serve, and at the same time minimises overhead movement of lifts in order to service requests. Lastly, the external entities: *doors* and *lift_sensors* provide status information to the corresponding lift controller.

The process objects inside a *lift_controller process* are:

- Data server *lift_position* accepts input from the external entity *lift_sensor*. Whenever requested, it provides the lift position information to the control process object *elevator_movement_control*, and a control process object *request_interpreter*.

- *Outside_request_filter* is modeled as a control process object. Its main function is to receive requests from the *request_allocator* and obtains movement information from the *movement_table*. It subsequently sends processed requests to *request_interpreter*.

- *Request_interpreter* is also a control process object. It receives requests to go to particular levels from two sources: the *outside_request_filter* - requests from outside the lift, and *panel_sensors* - request from inside the lift. *Request_interpreter* obtains current lift position information and determines which requests to be placed onto the *request_queue*.

- *Request_queue* stores requested level numbers.

- *Elevator_movement_control* obtains the next request to service from the *request_queue*. It determines the movement direction of the lift. If a change in direction is required, it sends an *update* message to the *movement_table*. This control process also tells the *door_manager* when the requested level has been reached or not. It does this by comparing the requested level with the current lift position.

- *Door_manager* controls the opening and closing of the lift door.

The decomposition of the two processes: *outside_request_filter* and *request_interpreter* are shown in Figure 3.12.

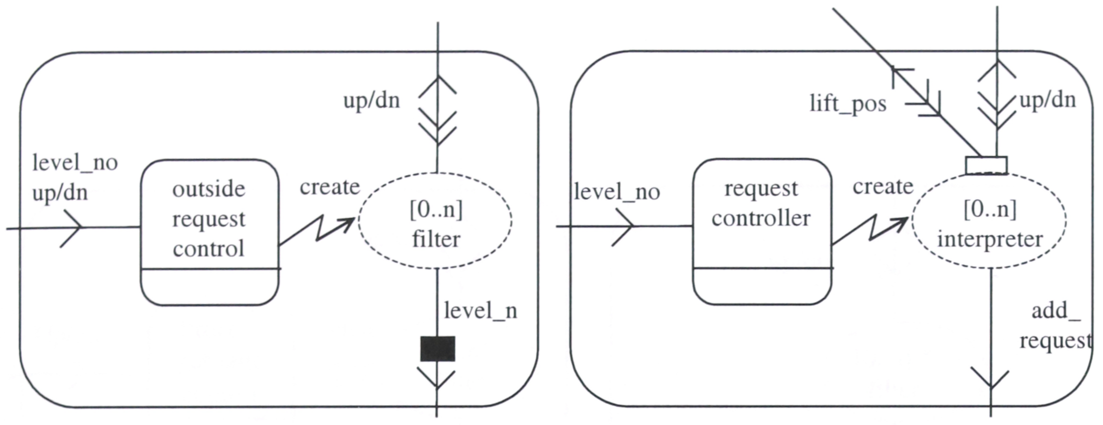


Figure 3.12. Decomposition of 'Outside Request Filter' and 'Requests Interpreter'

In both of these process objects, the level numbers (and up or down signals in the case of *outside_request_filter*) are received by the internal control processes, which then create dynamic worker processes to carry out the work. The information such as *level_numbers* and *up* or *down* signals are passed to corresponding instances of dynamic worker processes upon creation.

The *filter* process inside the process *outside_request_filter* simply checks whether the request is going up or down. It also obtains current movement from the movement table. If the request is in the same direction as the *lift_movement*, then the request is sent to *request_interpreter* which subsequently places the request onto the queue. If however, the request is in the other direction, the filter process waits till the lift movement has changed before sending request to the *interpreter*.

Within the process *request_controller*, each instance of the interpreter checks current *lift_movement* and *position* in order to decide if the request can be serviced without changing lift movement direction. Thus adding the request to the *request_queue*. Both requests from outside and inside the lift are serviced.

Both *filter* and *interpreter* processes die from *aging*.

Figure 3.13 illustrates examples of alternative designs where the filter process exits the system through either assassination or suicide.

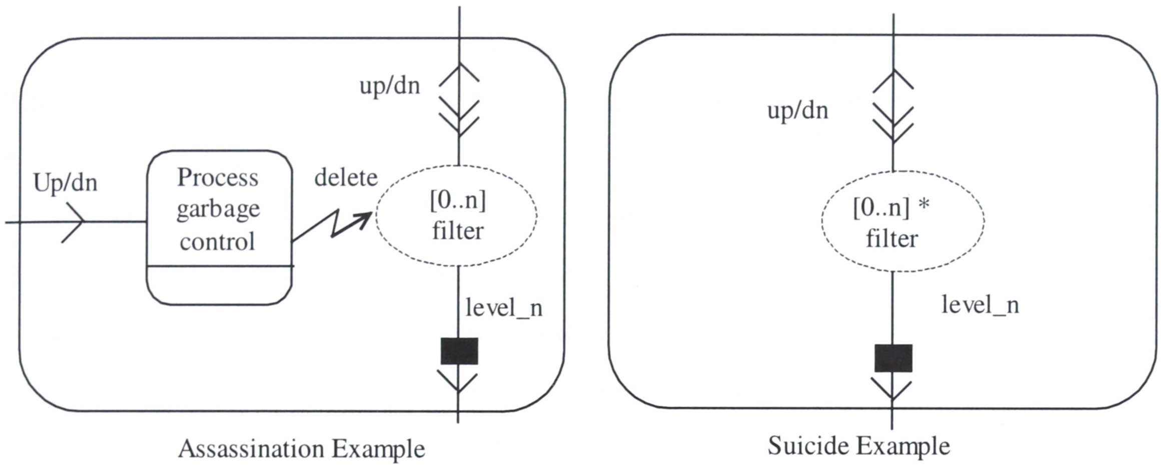


Figure 3.13 Examples of Assassination and Suicide

3.4 Conclusion

In this chapter, the Dynamic PARSE Process Graph Design Notation has been presented. The Dynamic PARSE Process Graph Notation has the following features:

- platform independent
- scaleable - encourages component-based design
- hierarchical - to manage system complexity
- user-friendly - graphical notation is simple to use
- rich set of process object types - enable architects to capture client-server relationships easily
- rich set of communication types
- path constructors encourages explicit specification of concurrency
- dynamic properties such as process creation/deletion and communication reconfiguration can be captured

Once the architectural properties have been captured using PARSE-D, the designs can be subjected to analysis and verification. In chapter 4, the formal semantics of Dynamic PARSE Process Graph Notation will be presented. This forms the basis of automated design verification.

4. Dynamic PARSE Analysis/Verification Methodology

4.1 Introduction

In this chapter, the Dynamic PARSE Design Analysis and Verification process will be described. In order to facilitate design analysis, Dynamic PARSE designs are translated into corresponding formal models. These mathematically sound models then form the basis for design analysis and verification.

Firstly, the translation scheme of Dynamic PARSE Process Graph Notation to the Petri Net formalism will be presented. Then, the translation scheme for Dynamic PARSE to the π-Calculus formalism will be described. A comparison between the use of Petri Nets and π-Calculus will then be presented.

4.2 Translation Scheme

4.2.1 Translating Dynamic PARSE Designs to Petri Nets

Introduction and Past Work

Petri Nets are an important mathematical model for the analysis of systems with interacting concurrent components. Petri Nets have been used previously as the auxiliary tool for PARSE design analysis [Jelly95]. Once the corresponding skeleton Petri Net model has been derived from a PARSE design, system properties can be analysed, and knowledge gained about the system can subsequently be used to revise the design. Some system properties that are desirable to be analysed include liveness and reachability properties.

It is thus important to devise a set of rules for the translation of PARSE design components to corresponding partial Petri Net models. Initial work in this area has been carried out [Jelly95][Pateman95] where mappings for the four communication types, basic programming structures such as sequence, selection and iteration, non-deterministic constructors and concurrent constructors have been described and explained. There was also a substantial case study carried out [Gorton96]. However, there has not been any work carried out on deterministic constructors, nor dynamic features such as those present in PARSE-D [Liu96].

In the following, the transformation from PARSE design elements to Petri Net models will be presented, followed by the transformation rules for Dynamic PARSE to Petri Nets. In order to model re-configurable system structures such as those commonly designed using Dynamic PARSE, it is necessary to use a higher-level Petri Net known as Self-Modifying Nets, proposed by Valk [Valk77].

PARSE to Petri Net

Here, the mappings for the four communication types are described. This is essentially the same as those presented in [Jelly95] [Pateman95] and [Gorton96]. In addition, the mapping for the three types of path constructors are shown.

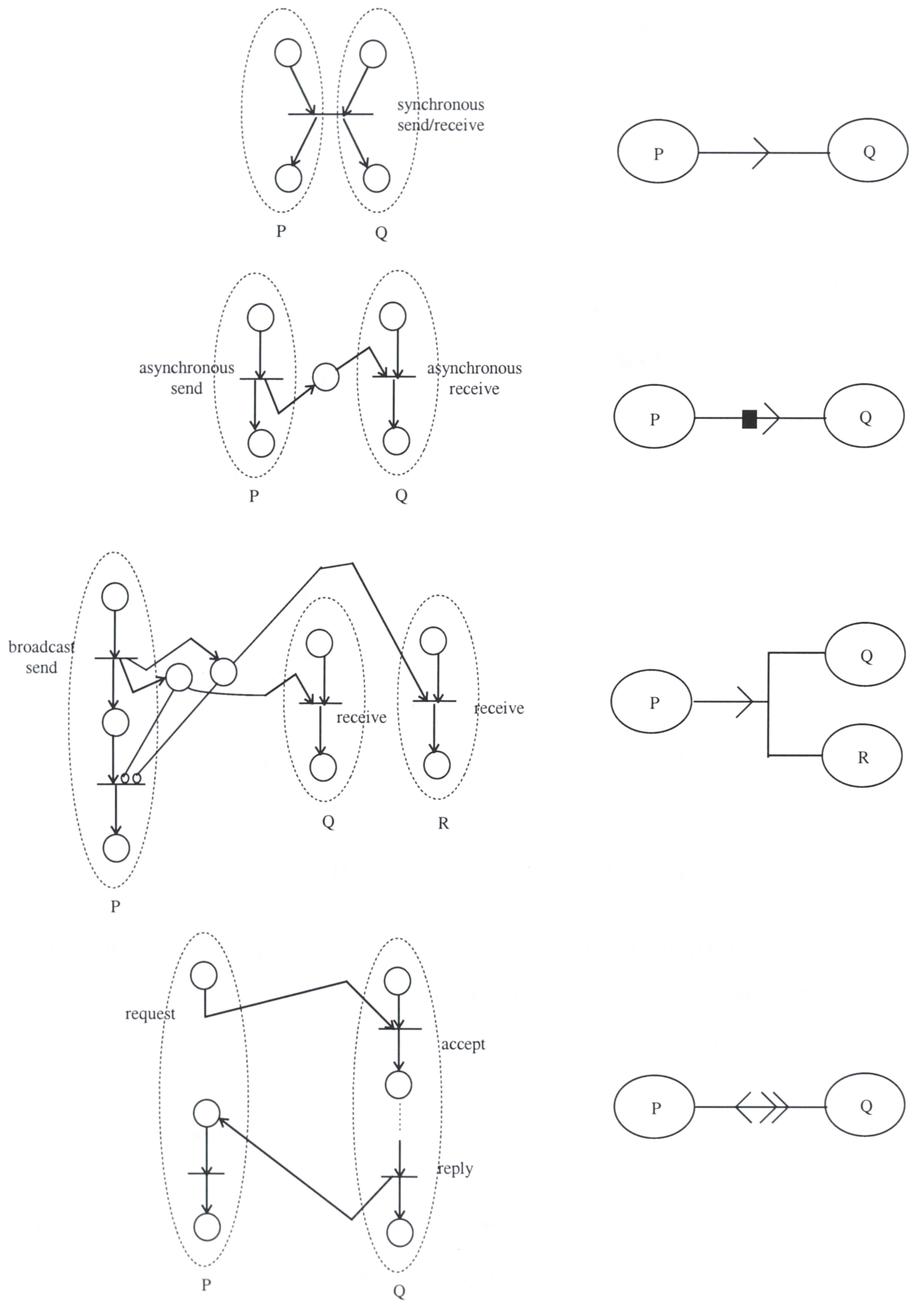


Figure 4.1. Petri Nets Model: Four Communication Types

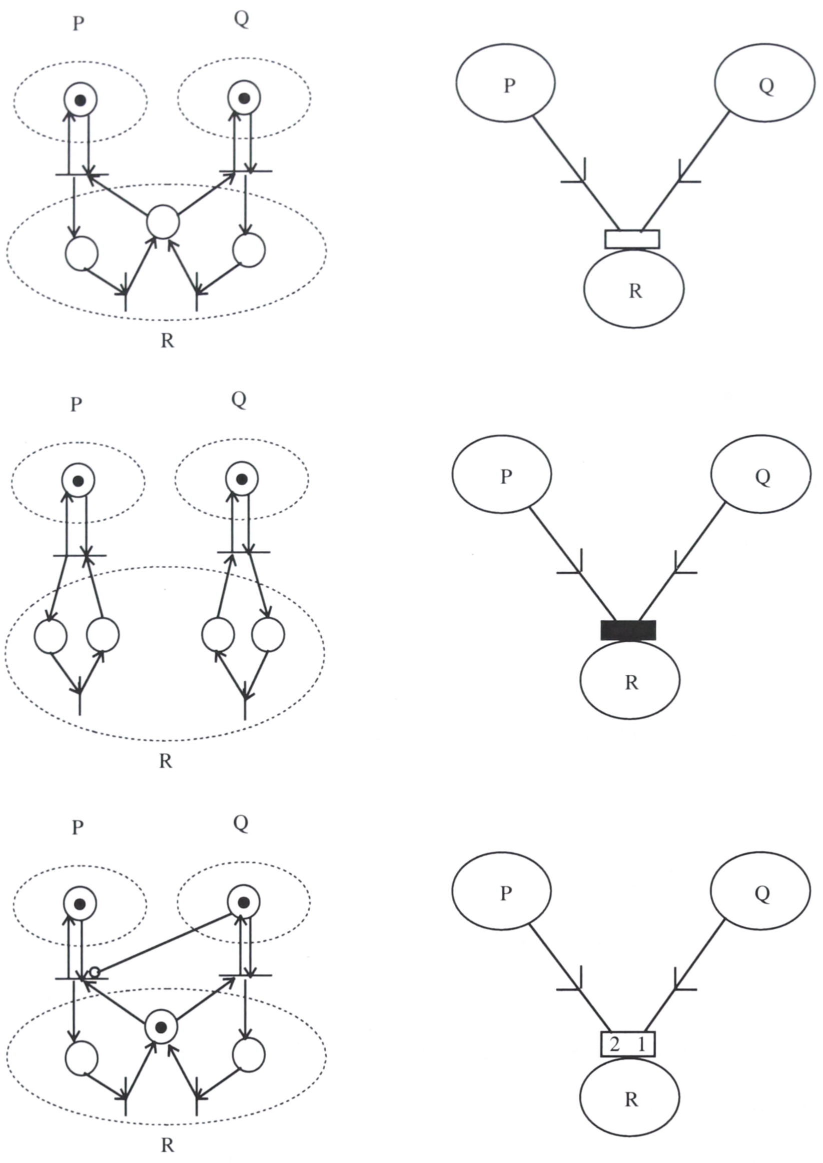


Figure 4.2. Petri Nets Model: Path Constructors

Basic Petri Nets are not expressive enough to model deterministic path constructors. The inhibitor arc (represented by an arc with a small circle at the end) is necessary to model the prioritised choice behaviour present in the deterministic path constructor.

PARSE-D to Self-Modifying Net

PARSE-D supports the design of systems with evolving system structures. To model systems designed using PARSE-D, Self-Modifying Net [Valk77] is used. Figures 4.3 and 4.4 present two example mappings.

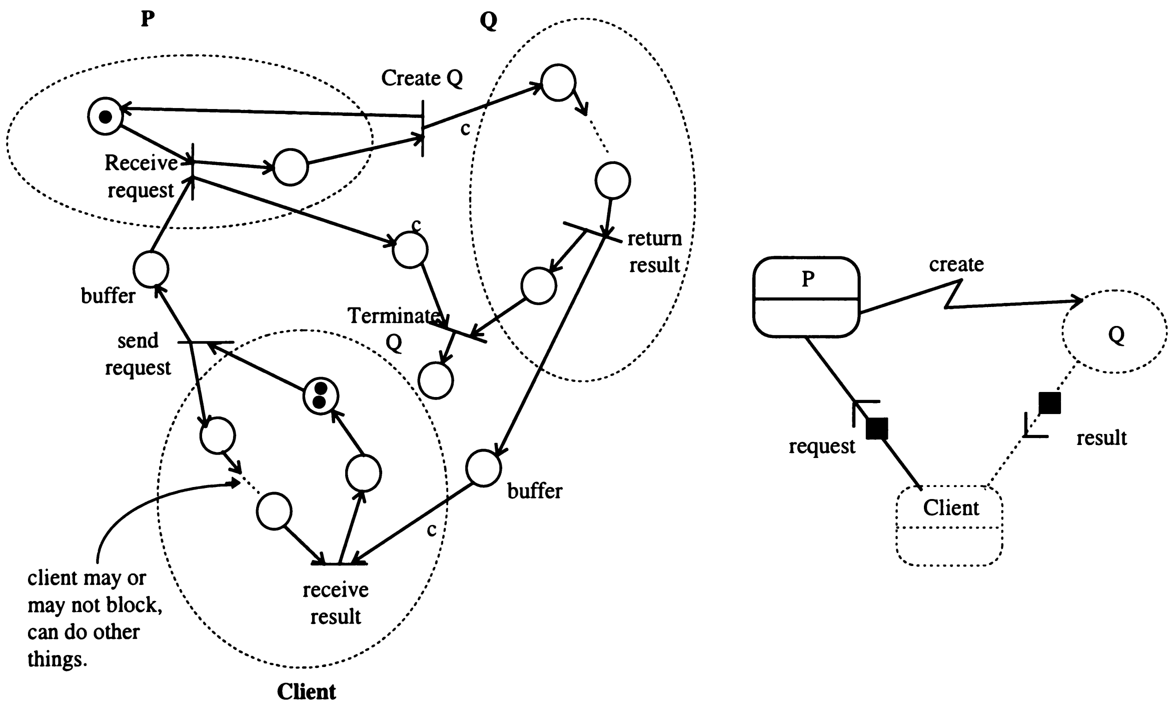


Figure 4.3. Self-Modifying Nets Model: Creating Worker Processes

For the *Creating Worker Processes* example:

- tokens are threads of control: initially, tokens are present in *P* and *Client* only, since they are the only active objects in the system at start up time;
- the tokens in *Client* means there are 2 clients active in system;
- place *c* counts the number of active *Q* processes in system at any time;
- when place *c* contains zero tokens, there are no active *Q* processes in the system. To reflect this change in the system, the two arcs labeled with *c* are disabled, thus separating the sub-net representing process *Q* away from the main net representing processes *P* and *Client*. Since the sub-net representing *Q* would have no token, it can be ignored;
- the dynamic *result* path is also appropriately modeled via the self-modifying arc.

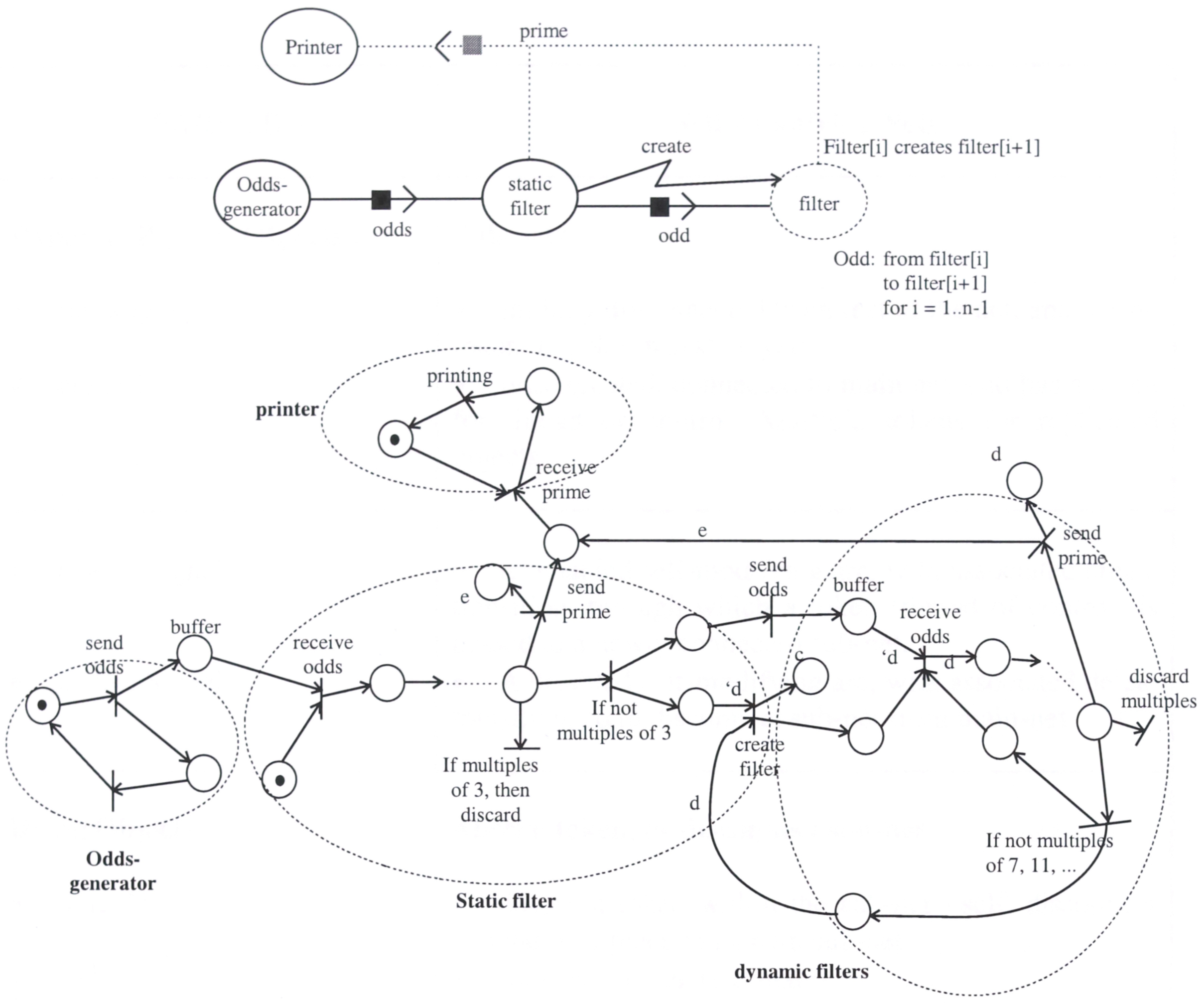


Figure 4.4. Self-Modifying Nets Model: Sieve of Erastosthenes Problem

For the *Sieve of Erastosthenes* example:

- Initially, 3 tokens in the processes: *printer*, *odds_generator*, and *static_filter*, representing that these are the 3 initial active processes.
- Place *d* is used to provide information regarding the last prime number being generated. The self-modifying arcs *d* and *'d* are correspondingly enabled and disabled.
- Place *d* is used for 2 purposes: for the creation of a new filter process, and for passing an odd number to the next filter in the pipeline.
- Place *e* is used to record if the static filter has generated a prime number or not. The self-modifying arc labeled with *e* thus represents the behaviour of a dynamic communication path *prime*.

Table 4.1 presents the mappings between PARSE-D design notation components and Self-Modifying Nets.

PARSE-D	Self-Modifying Nets
Dynamic Process Objects • initially • created	**Sub-nets** • initially disconnected from rest of the net, and has no token, i.e. Not in system yet! • once created, connected to main net, and has a token for thread of control. Multiple tokens for replicated objects.
• Create Signal • Delete Signal	• an enabled self-modifying arc, with associated create transition, through which a token (thread of control) is passed to a newly connected sub-net. • A disabled self-modifying arc, with associated delete transition, so to disconnect sub-net from main-net.
Dying Objects • Suicide • Assassination • Aging	**Absorb token, or disconnect sub-net** • absorb a token within the sub-net itself, disconnect by reading a place in its own sub-net • same as the delete signal • once a token has traversed through all places, the token is absorbed. Use self-modifying arc to disconnect sub-net when no tokens left.
Transactional Communication Paths • synchronous send/receive • asynchronous send/receive • broadcast • bi-directional	**Self-Modifying arcs** otherwise same as for static communication paths.

Table 4.1. Summary Mapping of PARSE-D Components To Self-Modifying Nets

4.2.2 Translating Dynamic PARSE Designs to π-calculus

Monadic π-calculus

The π-calculus is a calculus for describing and analysing concurrent systems with evolving communication structures. The simple monadic form of π-calculus is used here. π-calculus is based around the notion of naming. A system in the π-calculus is a collection of independent processes that communicate via channels that in turn are referred to by names. Names are the most primitive entities in the calculus, and have no structure. There are infinite numbers of names, generally represented by lower case letters.

Processes can be built up from names in the following ways:

Action term ::= $\overline{x}\langle y\rangle.P$	Outputs the name *y* along the channel named *x* then executes *P*.
$x(y).P$	Input a name, call it *y*, along the link named *x*, and then execute *P* (binds all free occurrences of *y* in *P*).
terms ::= $P1 + \ldots + Pn$	Choice operator, alternative actions, execute only one of *Pi*.
$P1 \mid P2$	Composition (both + and | commutative and associative). *P*|*Q* means that *P* and *Q* are concurrently active. They can act independently, but can also communicate.
$(\nu y)P$	Restriction operator. Introduces a new name *y* with scope *P* (binds all free occurrences of *y* in *P*).
$!\,P$	Replication operator. Provides many copies of *P*. *!P* means *P* | *P* | ... as many copies as you like. It satisfies *!P* = *P* | *!P*.
0	Dead process.

Communication in the calculus is expressed by the following reduction rule:

$$COMM: (\ldots + x(y).P1\ldots) \mid (\ldots + \overline{x}\langle z\rangle.P2 + \ldots) \rightarrow P1\{z/y\} \mid P2$$

Sending z along channel x reduces the left hand side to $P1 \mid P2$ with free occurrences of y in $P1$ replaced by z. Notice that $\{z/y\}$ has the same meaning as α-reduction in λ-calculus, which basically means replace free occurrences of y with z.

PARSE-D to π-Calculus

Process Objects: Similar to the classical process algebra model, a system is composed of a collection of interacting agents, where each agent is defined by the actions it can perform or as a composition of smaller agents. Hence, the agents in π-calculus may be used to represent process objects in PARSE, with basic message passing modeled via the input and output bindings such as the action terms $x(y)$ *and* $\overline{x}<y>$.

-- Communication Paths --

Synchronous: In π-calculus, a reduction between a pair of input binding and an output binding can be used to synchronise communication between two agents. The reduction rule discussed in section 4.2.2 (previous page) is synchronous in nature.

e.g. $P = \overline{x}.P$ *and* $Q = x.Q$

Here, P does a blocking send along communication path x, i.e. P cannot send messages to Q until Q is ready to receive.

Asynchronous: in asynchronous communication, the sender does a non-blocking send:

$$P = \overline{x}.0 \mid P$$

Hence, in a more complex environment, P may resume doing other work, rather than waiting for messages along x to be successfully received.

Bi-directional: a bi-directional communication typically models the request-reply type of communication protocol. In π-calculus, we model it using two uni-directional synchronous communication bindings. However, we need to make sure that the *request* is sent earlier in time than *reply*.

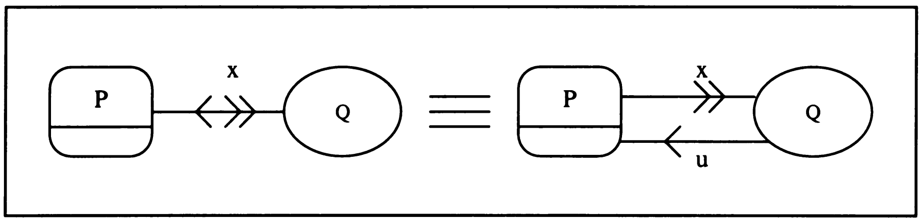


Figure 4.5. Bi-directional Communication

A single bi-directional communication can be modeled as:

$\bar{x}<u>.u(z).0 \mid x(u).\bar{u}<v>.0$ *or* $\bar{x}<u>.u(z) \mid x(u).\ \bar{u}<v>$

where *x* is the forward request path, and *u* is the reply path.

A continuous bi-directional communication between processes *P* and *Q* can then be modeled as:

$P(x,u) \equiv \bar{x}<u>.u(z).P(x,u)$

$Q(x,v) \equiv x(u).\bar{u}<v>.Q(x,v)$ *v is the desired/computed result*

and thus the system is:

$system \equiv P \mid Q$

where processes *P* and *Q* are executed concurrently.

Broadcast:

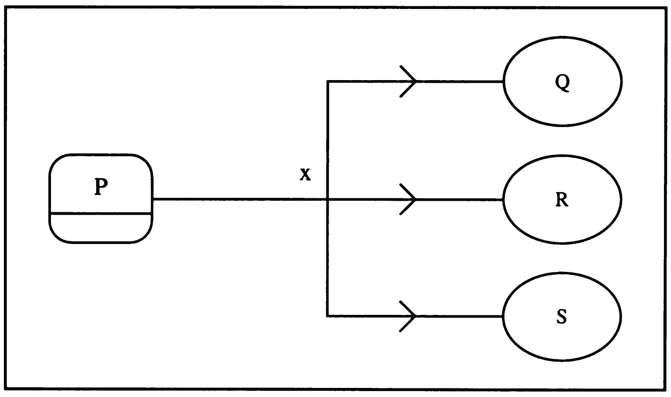


Figure 4.6. Broadcast Communication

Consider this:

$P \equiv \bar{x}<y>.P$

$Q \equiv x(y).0$

$R \equiv x(y).0$

$S \equiv x(y).0$

$System \equiv P \mid Q \mid R \mid S$

Upon receiving *y* from channel *x*, processes *Q*, *R*, *S* becomes null processes 0 which can not continue receiving copies of *y*. Hence *Q*, *R* and *S* each receives one copy of *y*.

Further, *P* does not need to know how many copies of *y* to send out. *P* is repeated to output unbounded copies of *y* through channel *x*. The system is terminated when no active processes exist to receive *y* through *x*.

Although there is no primitive broadcast operation in π-calculus, we can simulate it via a combination of the primitives presented in this section. We may view the above model of broadcast communication as an atomic operation, thus achieving the purpose of design verification and deadlock detection using the corresponding formal model.

-- Communication Path Constructors --

In PARSE, the order of handling of multiple input to a process object can be explicitly specified using path constructors. The various types of path constructors include deterministic, non-deterministic, and concurrent.

Non-deterministic

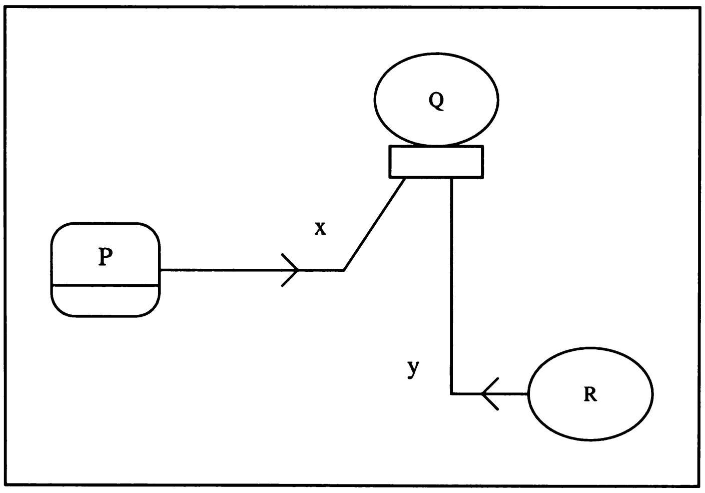


Figure 4.7. Non-deterministic Input Ordering

Consider Figure 4.7. Process *Q* handles the incoming messages in a non-deterministic order. The choice operator in π-calculus '+' is non-deterministic:

$$P = \overline{x}<v>.P$$

$R = \overline{y}<w>.R$

$Q = x(v).Q + y(w).Q$

In this case, it is non-deterministic whether message along path *x* will be handled first by *Q*, or message along path y takes precedence.

Concurrent

In this case, the receiver process is always composite, and there are internal primitive processes that may handle the multiple inputs separately/independently.

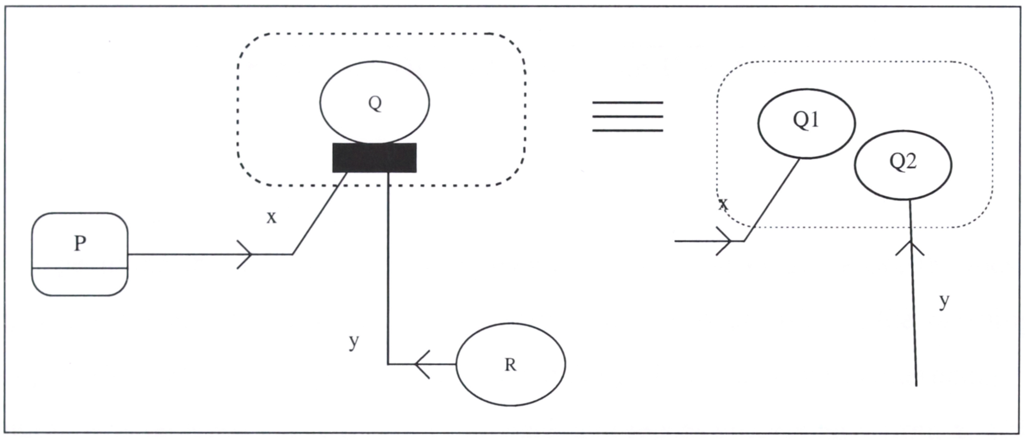


Figure 4.8. Concurrent Input Ordering

The corresponding π-calculus model for Figure 4.8 is as follows:

$Q = Q1 \mid Q2$

$Q1 = x(v).Q1$

$Q2 = y(w).Q2$

and the definitions of *P* and *R* remains the same as the case for non-deterministic input handling.

Determinstic

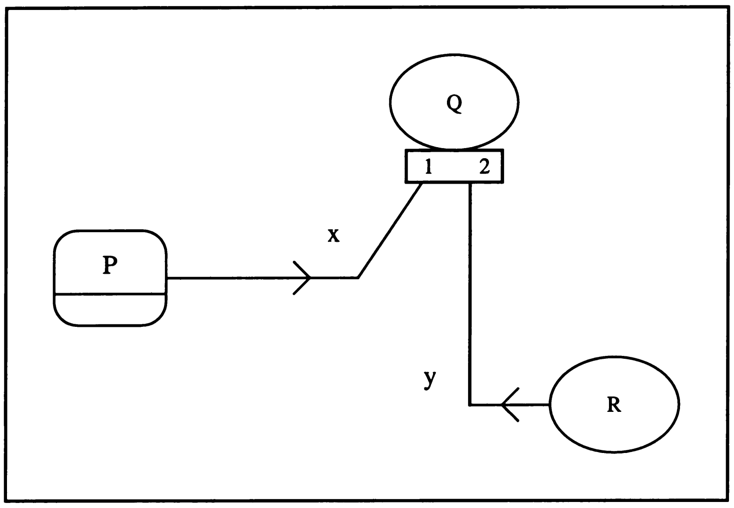


Figure 4.9. Deterministic Input Handling

Consider:

$$Q = x(v).y(w).Q$$

this is specifying that the message from channel *x* will be received before the message along *y* is received. Hence, it implicitly specifies some sort of order in the handling of messages. However, it also specifies that the handling of the messages is alternating, that is, unless *w* is received via *y*, *Q* cannot go back and repeat processing *v* from *x*. This is clearly an undesirable behaviour.

If we model it this way:

$$Q = x(v).y(w).Q + y(w).Q + x(v).Q$$

The first choice $x(v).y(w).Q$ is for when both input from channels *x* and *y* arrive together, then *x* is handled first before input from *y*, since *x* is the higher priority channel. The second choice $y(w).Q$ and the third choice $x(v).Q$ are for when the inputs from *x* and *y* arrive at different times, then whichever input has arrived, it is handled by *Q*.

-- Hierarchical Construction --

The Dynamic PARSE design notation supports hierarchical construction of software components.

When using π-calculus for correctness verification, the description of the high-level (composite) object is simply the total of all its internal process objects and the behaviour of the higher-level object is the individual behaviour of the internal objects combined.

Figure 4.10 is a simple PARSE design which is constructed hierarchically:

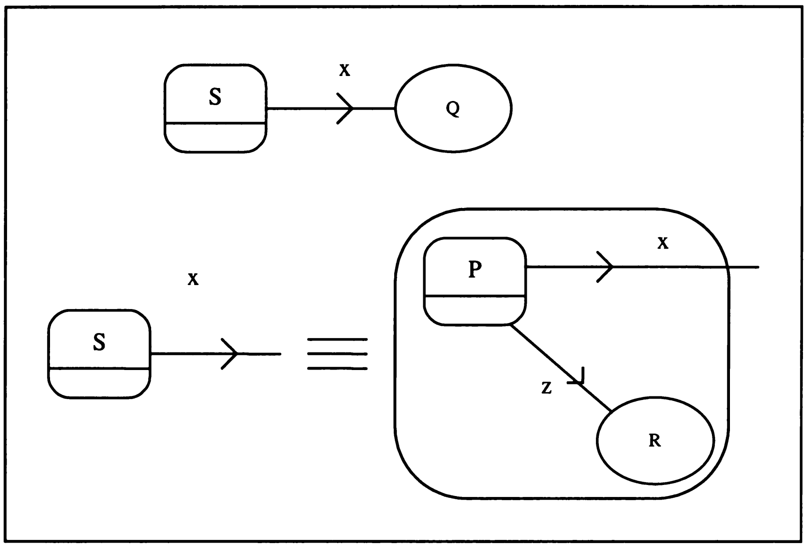


Figure 4.10. A Simple Hierarchically Constructed PARSE Design

The corresponding π-calculus description is as follows:

$System = (\nu\, x)(S(x) \mid Q(x))$

$Q(x) = x(y).Q(x)$

$S(x) = (\nu\, y,z,w)(P(x,y,z,w) \mid R(z))$

$P(x,y,z,w) = \overline{x}<y>.\overline{x}<w>.P(x,y,z,w)$

$R(z) = z(w).R(z)$

Notice the use of the restriction operator ν in the description of S. The name z is restricted so that its scope is within S. Hence, even if z occurs freely in Q, it is not the same z as that which is restricted to P and R.

-- Other Dynamic Process Structures --

Dynamic process creation can be modeled in π-calculus easily due to the basic idea of naming, and that all references to various objects can be passed just like names.

Consider the example in Figure 4.11 where process *P* creates a helper process *Q* to carry out work for a *client*.

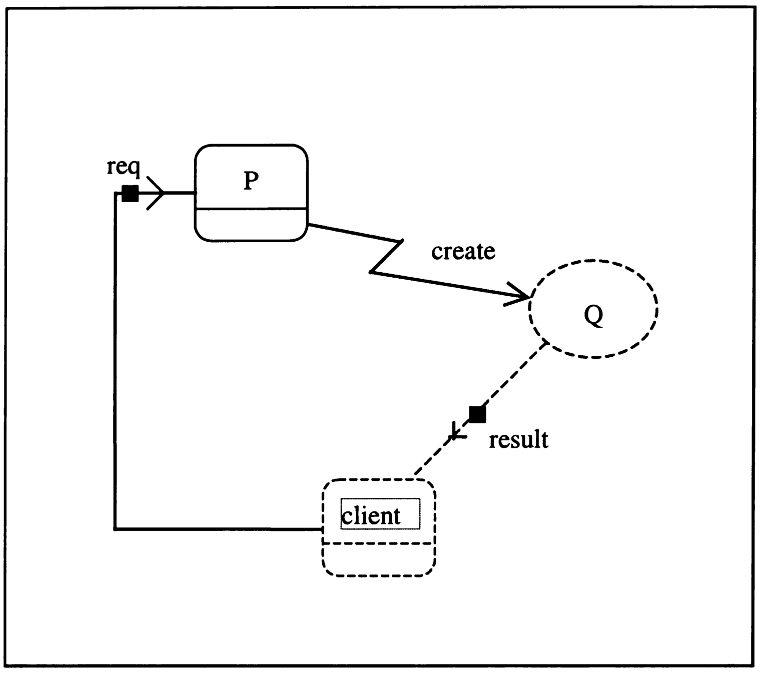


Figure 4.11. Dynamic Process Creation

The creation of process *Q* by process *P* can be modeled as in the following:

$$P(req,create) = req(result).\overline{create}<result>.(P(req,create)|\ (\hat{}r)(Q(r,create)))$$

where *P* sends the handle to the *Client* communication path to *Q*, namely *result*, and *Q* then subsequently uses to send reply of work done back to the *client*. Notice that *P* has the ability to create several copies of *Q* depending on the number of requests from *clients*. The process *Q* is defined as follows:

$$Q(r,create) = create(result).\overline{result}<r>.0$$

In this case, the helper process *Q* dies of *aging*, that is, it automatically exits the system once its work has been carried out. Process *Q* may also exit the system via the *assassination* method, where it terminates when receiving the signal *delete* from another process, possibly process *P*:

$$Q(r,create,delete) = create(result).\overline{result}<r>.Q(r,create,delete) + delete(die).0$$

Lastly, we need the client processes to fire off the request:

$$client = \overline{req}<r>.client$$

and the total system at the beginning can be modeled as:

$$system = client \mid P$$

to reflect the fact that process *Q* does not exist at compile time, but it may be created by process *P* during run-time.

-- Dynamic Communication Structures --

Dynamic communication structures occurring frequently in (for example) mobile systems can be easily modeled using π-calculus. Figure 4.12 is a PARSE design of an example of a dynamic communication structure. In this example, the control and access to a specific communication path is not held exclusively by one and only one process object, but is passed around between various different process objects.

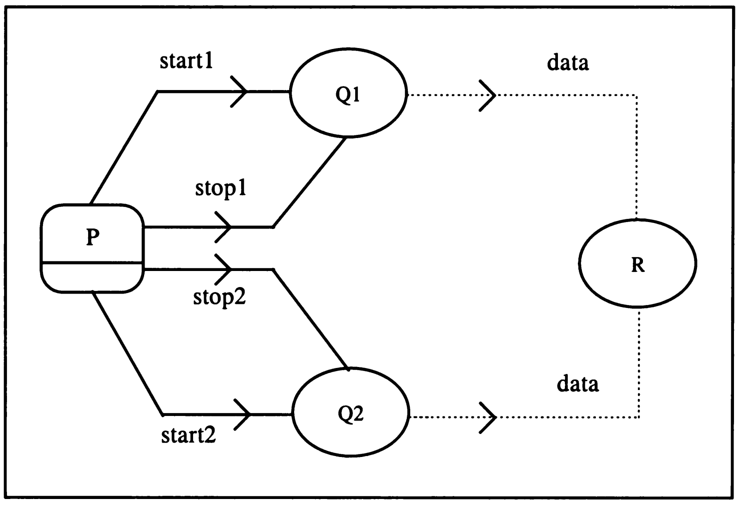


Figure 4.12. Dynamic Communication Structure Example

Here, the *data* communication path is dynamic, where process objects *Q1* and *Q2* have access to the *data* communication path at different times in order to pass information to process *R*.

The corresponding π-calculus expressions for this example is as follows:

$P1(start1, stop1, start2, stop2, data) =$

$\overline{stop1}.\overline{start2}<data>.P2(start1, stop1, start2, stop2, data)$

$P2(start1, stop1, start2, stop2, data) =$

$\overline{stop2}.\overline{start1}<data>.P1(start1, stop1, start2, stop2, data)$

$Q1(start1,stop1,data) = \overline{data}.Q1(start1,stop1,data) + stop1(s1).IdleQ1(start1,stop1)$

$Q2(start2,stop2,data) = \overline{data}.Q2(start2,stop2,data) + stop2(s2).IdleQ2(start2,stop2)$

$IdleQ1(start1,stop1) = start1(data).Q1(start1,stop1)$

$IdleQ2(start2,stop2) = start2(data).Q2(start2,stop2)$

$R(data) = data.R(data)$

SYSTEM = (^start1,start2,stop1,stop2,data)
(P1(start1, stop1, start2, stop2, data)
| Q1(start1,stop1,data)
| IdleQ2(start2,stop2)
| R(data))

Note: $\overline{data}$ in the above π-calculus expression is an output channel. This is a shortcut where the name of the output is not specified. This syntax is commonly used in various π-calculus literature [Milner91] [Milner92], and is also accepted by the automated π-calculus tool [Victor94].

Summary

Table 4.2 is a summary of mappings from Dynamic PARSE to π-calculus.

Dynamic PARSE	**Corresponding π-Calculus Expression**
A software architecture	A parallel composition of a set of interacting agents.
Process objects	Agents
Communication Paths (Message passing) · Synchronous · asynchronous · bi-directional · broadcast	Input/output bindings, i.e. action term such as <y> and x(y). · a pair of input/output bindings, e.g. ... <y>... | ...x(u)... reduction rule is synchronised · sender sends a non-blocking send, e.g. P = | P (analogous to asynchronous π-Calculus), or, insert a buffer agent between the coupled input/output binding. · 2 uni-directional synchronous communication bindings, where time ordering is strictly request before reply: e.g. .. <u>...u(v)...| ...x(z)... <y>... · can be simulated with a set of asynchronous uni-directional communication
Path constructors · non-deterministic · concurrent	Different ways of "composing" multiple communication paths so as to have different ways of ordering. · Choice operator is non-deterministic, e.g. P = x.P + y.P · Independent handling by different sub-components e.g. P(x,y) = P1(x) | P2(y), P1(x) = x.P1, P2(y) = y.P2

· deterministic (prioritised input handling)	· There is no prioritised choice in p-Calculus. However, for deadlock analysis and verification purposes, we simplify this to be a non-deterministic choice of all possible combination of choices, thus covering all possible situation for state exploration.
Dynamic Process Creation Process Deletion · aging · assassination · suicide	Newly spawned process is to be composed in parallel to the parent process. The name 'create' can be used as a handle to pass any other names (parameters, state info) to the newly created process. Process exits system by becoming null '0' process. · After all action terms, agent becomes '0', e.g. P = y. .0 · Explicit 'delete' signal sent to process to be deleted. e.g. P = … …., Q = … d…, agent P assassinates Q here · Upon receiving certain state information (name), process terminates itself by becoming the null '0' process
Transactional Communication Paths	Handles to communications paths are simple 'names'. Names can be freely passed around to different agents, thus achieving re-configurable communication structure.
Hierarchical Construction /modular encapsulation	Use of restriction operator n for hiding internal details of a module, e.g. S(x) = (nz)(P(x, z)|R(z)), where z is an internal communication path of the composite module S.

Table 4.2. Summary Mapping of PARSE-D Components to π-Calculus Expressions

4.2.3 Comparison between the π-Calculus and Petri Nets Approach

I. Mathematical Foundations

π-Calculus is a process algebraic approach, whereas Petri Nets is a state automata approach, which is based on bag theory.

The process algebraic and the state automata approaches are better than set-theoretic models such as Z, VDM, and B in describing the actual happenings during reconfiguration of a distributed system. While set-theoretic models capture the states of components as a result of (communication) reconfiguration, they are not equipped to describe the actual happenings during the reconfiguration. Both the process algebraic and state automata methods can better capture the interaction between process components, and can be 'transformed' or 'executed' to model behaviour of the system over time.

Both π-Calculus and Petri-Nets use some form of state-space exploration method (e.g. reachability trees, traces) for model checking.

II. Expressiveness of Modeling Elements

π-calculus enables the modeling of concurrent and distributed processes where the system structure is dynamic. It is based on the notion of naming and the passing of those names between processes in the system, to enable dynamic communication, or message passing amongst entities. Complex, powerful features can be expressed in π-calculus precisely and succinctly.

The basic place-transition Petri Net is not expressive enough to model distributed systems with dynamic features. An extension of Petri Net, namely Self-Modifying Net must be used in order to model dynamic structural changes in a system.

III. Pragmatics - Ease of Use

The use of π-Calculus for systems modeling involves a substantial training and learning period at startup. In general, software engineers are reluctant to use formal algebraic methods to model systems [Saiedian96].

Petri-nets can have textual representation for tool machine analysis, but also importantly, the graphical representation of concurrency is intuitive to use.

IV. Hierarchical Composition/Decomposition

π-Calculus models support true hierarchical structuring. Composition is carried out via well defined mathematical operators, + (choice composition) and | (parallel composition), coupled with encapsulation, and the λ operator for hiding in modules.

Petri Nets do not support true hierarchical composition [Valmari96]. Different levels of designs are firstly flattened in order to generate a single large Petri Net model for the system. This has implications in limiting the use of composition reachability analysis [Cheung94][Russo97].

V. Automatic Verification Tool Support

For the basic Petri Net, the formal framework has been thoroughly researched. There are various mature Petri Net tools that support the construction, and subsequent verification of Petri nets. An example of a widely used Petri Nets tool is Design/CPN [Jensen92]. However, for Self-Modifying Nets, there is currently no known tool support. It has been shown that the reachability problem is undecidable for Self-Modifying Net [Valk78]. In fact, it has been shown that extended Petri Nets with the ability to test for zero allows a Petri Net to simulate a Turing machine. Thus, a Petri Net with inhibitor arcs can model any system. As a consequence, almost all analysis questions for Petri Nets become undecidable, since they are undecidable for Turing machines. Hence, the Self-Modifying Net is a good theoretical framework for modeling PARSE designs, but subsequent analysis of the resultant Self-Modifying Net can only be done manually, and only to a certain extent.

Model checking tools for π-Calculus also suffer from the same state explosion problem. However, this problem is alleviated in two ways. Firstly, the use of compositional reachability analysis is possible with π-Calculus models, hence an analysis of the system design can be carried out on isolated modules, and if required, integrated analysis with black-box components that have been proven correct can be carried out. Secondly, the π-Calculus model checking tool (Mobility Workbench) employs heuristics which reduce the search space during model checking. The details for the *on-the-fly* algorithm can be found on [Victor94].

In short, π-Calculus is supported by the Mobility Workbench, and there is no tool support for Self-Modifying Nets.

VI. Future Work

There is an abundance of theoretical work in the area of π-Calculus equivalence. The equivalence theory, once mature, can be used to aid the refinement of specifications [Morgan90], and bridge the gap between design and implementation. There is much scope in this area of research.

For Petri Nets, the translation to program code is not possible. The refinement of Petri-nets does not give rise to interesting results.

4.3 The Dynamic PARSE Design Analysis and Verification Process

The general approach to the development of the Dynamic PARSE Design Analysis and Verification process involves the following steps:

- choose an appropriate formalism
- devise mapping from Dynamic PARSE design features to the formalism chosen
- reason about system by analysing the formal model

Dynamic PARSE designs can be analysed for the presence of structural deadlocks [Birkinshaw95]. In a Dynamic PARSE design, a structural deadlock occurs at an architectural level when there is circular dependency amongst process objects. In π-Calculus terms, deadlock occurs when a system of collaborating agents is equivalent to a dead process. That is, when the system cannot proceed to do any useful work. For an in-depth discussion on π-Calculus' equivalence theory, please refer to [Sangiorgi96].

The Dynamic PARSE Design Analysis and Verification process works in two ways:

Process 1: Design analysis process for a single design - design refinement aid

A designer would typically go through the following process in developing a software architecture using the Dynamic PARSE method:

1. devise an initial software architecture design using the Dynamic PARSE process graph notation
2. transform the Dynamic PARSE design to the corresponding formal model (e.g. π-Calculus)
3. analyse the formal model for any design faults (e.g. deadlock checking)
4. revise the initial Dynamic PARSE design to eliminate design fault
5. repeat steps 2-4 to refine the software architecture design.

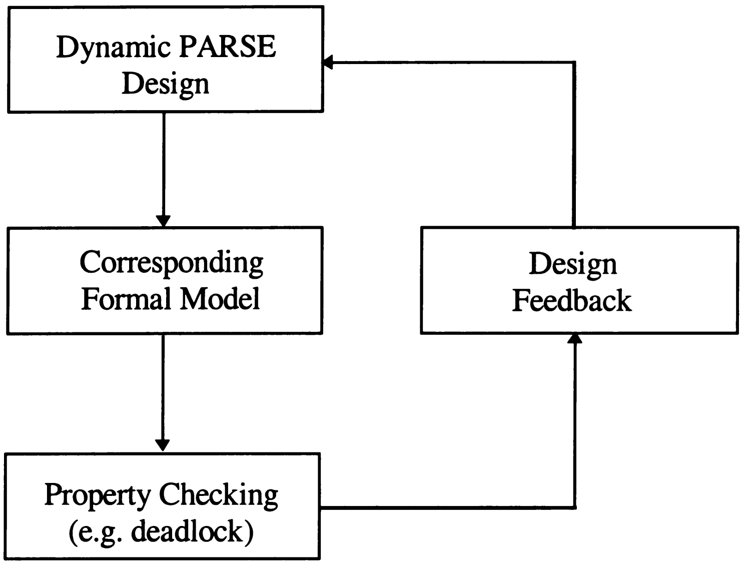


Figure 4.13. The Dynamic PARSE Design Analysis/Verification Process for Single Designs

Process 2: Design analysis for multiple designs (design alternatives) - design decision aid.

The Dynamic PARSE method can also be used as a design decision aid by taking the following steps:

1. devise multiple/alternative designs using the Dynamic PARSE process graph notation
2. transform the alternative Dynamic PARSE designs to their corresponding formal models.
3. analyse the various formal models for design faults.
4. based on the feedback from step 3, choose the best design alternative.

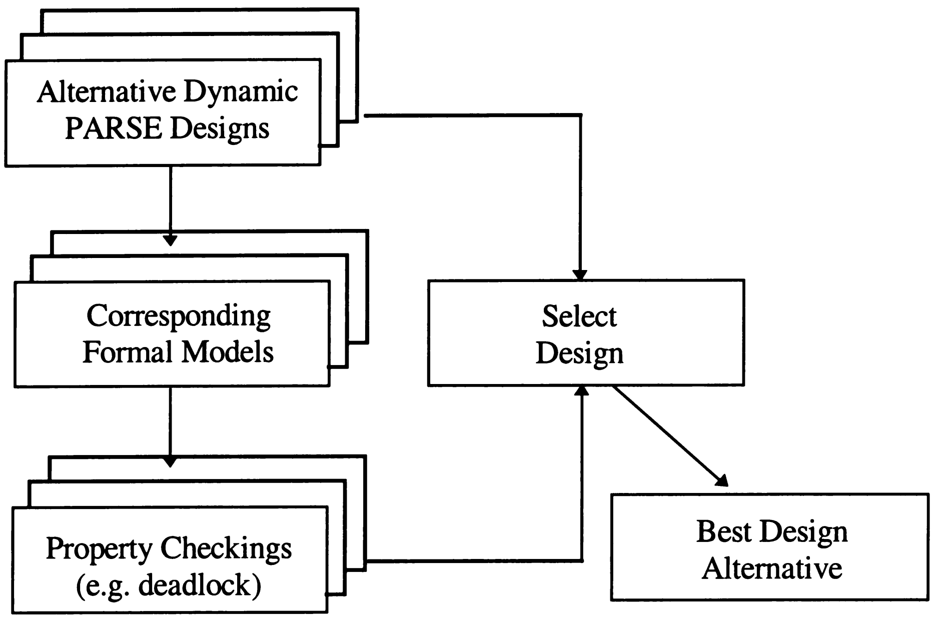

Alternative Dynamic PARSE Designs
Corresponding Formal Models
Property Checkings (e.g. deadlock)
Select Design
Best Design Alternative

Figure 4.14. The Dynamic PARSE Design Analysis/Verification Process for Multiple Design Alternatives

Hybrid approaches can also be taken.

- Multiple design alternatives can be analysed first. Once a particular design has been selected, it can be further refined iteratively.
- All designs can firstly be refined separately using Process 1, and then a single design can be chosen from analysing all refined design alternatives.

The Dynamic PARSE Design and Analysis processes are supported by the PARSE-DAT tool. Chapter 5 details this supporting environment.

4.4 Conclusion

Additional confidence in the validity and correctness of the Dynamic PARSE process graph design notation is gained via π-calculus and Petri Net modeling.

We have chosen π-calculus as the formalism to support design analysis and verification in Dynamic PARSE over the Self-Modifying Net. This decision was made according to comparisons between them in the areas of expressiveness, usability of modeling language, tool support and possible future works.

However, π-calculus is a complex formalism that has a steep learning curve, and often software professionals do not have the time or resource to master its use. Hence, tool support for this verification should be constructed to hide the details of π-calculus from software engineers.

By using the π-calculus formalism to model Dynamic PARSE designs, we also have increased understanding of the role and nature of module interconnection languages [Rice94].

All of the π-calculus definitions in this chapter have been checked for correctness and deadlock freedom using the Mobility Workbench [Victor94], which is an automated tool for manipulating and analyzing systems described in π-calculus.

By adopting the Dynamic PARSE Design Analysis and Verification methodology, software developers may reduce risks involved in large-scale software development projects. Design alternatives can firstly be explored through analysis according to certain properties, before an investment is placed on a single design. Chapter 5 will describe the PARSE-DAT environment which supports the design and analysis stages of the Dynamic PARSE methodology. This supporting environment will simplify the use of the Dynamic PARSE methodology by automating the analysis process.

5. Dynamic PARSE Design Analysis Tool

5.1 Introduction

For a software engineering methodology to be widely adopted, it is important to have accompanying tool support. A Computer Aided Method Engineering (CAME) tool and or a Computer Aided Software Engineering (CASE) tool should have the following features [Quatrani98]:

- easy to use graphical editing environment, which supports:
    - click and drop diagram construction,
    - consistency checks,
    - syntax and semantic validation (of design notation),
    - navigation,
    - printing facility,
    - documentation,
    - repository and multi-user support.
- provision for abstraction of underlying formal model
- support for design analysis
- provision for seamless transition between development stages
- steps involved in the use of the tool must reflect the development process as described by the methodology.

There are several ways to achieve CASE and CAME tool support for a new software engineering methodology [Gray97]:

1. Use existing CASE tools that have been built for a different, but similar methodology.

2. Use general-purpose application software such as word processors and drawing packages.
3. Construct full-featured custom tools using metaCASE products.
4. Construct custom tools using other implementation technologies not specifically for CASE tool constructions. For example, hand coding in 3GLs.

We have chosen to use a metaCASE tool (method 3) in constructing a CASE tool that supports the Dynamic PARSE design and analysis methodology. These metaCASE tools are specialised application generation environments specifically intended for the construction of new CASE/CAME tools. Naturally, resultant CASE/CAME tools will be more powerful than simply reusing existing CASE/CAME tools (method 1) or from using general purpose packages (method 2). At the same time, less tool development time and resources are required, as we are not building CASE/CAME tools from scratch as in method 4 [Gray97].

PARSE-DAT (PARallel Software Engineering – Design Analysis Tool) is an integrated environment that enables the design and analysis of distributed software architectures. PARSE-DAT supports the Dynamic PARSE Design and Analysis/Verification methodology by providing a design tool for the software design stage (PARSE-DT), as well as supporting the translation of graphical designs into corresponding π-Calculus formalism expressions, for analysis and verification purposes (PARSE-AT). Figure 5.1 illustrates the iterative design process, with feedback from the analysis/verification result obtained from the tool.

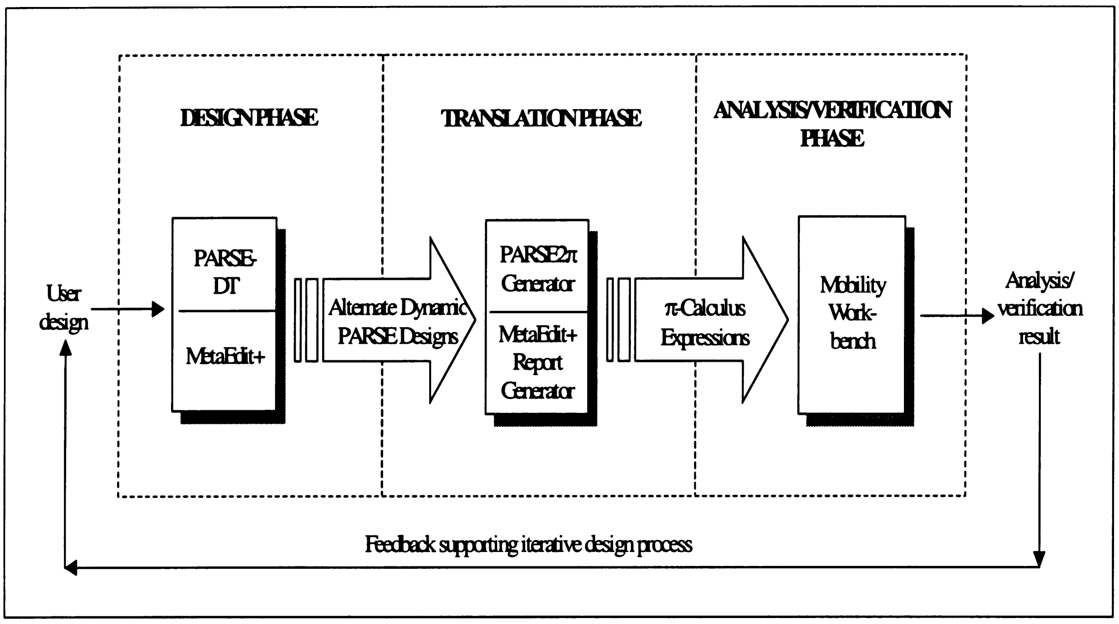

DESIGN PHASE
TRANSLATION PHASE
ANALYSIS/VERIFICATION PHASE
User design
PARSE-DT
MetaEdit+
Alternate Dynamic PARSE Designs
PARSE2π Generator
MetaEdit+ Report Generator
π-Calculus Expressions
Mobility Work-bench
Analysis/ verification result
Feedback supporting iterative design process

Figure 5.1. Dynamic PARSE Design and Verification Methodology

This chapter firstly presents the metaCASE approach to the implementation of the PARSE-DAT design and analysis/verification environment. Then, the PARSE-DAT design environment and the PARSE-DAT analysis/verification environment will be described. Sample use of the integrated environment will be given through a sample design constructed and formally analysed in PARSE-DAT.

5.2 The MetaCASE Approach

5.2.1 MetaEdit+ Overview

MetaEdit+ [Metacase96a] is a metaCASE tool that can be customised to allow the construction of multi-user CASE tools to support different software engineering methodologies. The specification of methods in MetaEdit+ is managed with the Method Workbench method engineering toolset. This includes tools that can be used to describe the methods to be supported in MetaEdit+.

The following is a list of features of MetaEdit+:

- Multi-user - several users can concurrently operate on the same repository
- multi-tool - different tools provide a different view of the same object

- multi-method - the environment provides mechanisms for method integration and consistency checking
- multi-form - the environment provides several representation formats for the same design object
- multi-platform - MetaEdit+ is platform independent both for the environment and the data.

Further, MetaEdit+ provides the following four families of tools:

- Environment management tools
- Model editing tools
- Model retrieval tools
- Method management tools

Table 5.1 taken from [Metacase96c] details the functionality of the various tools.

Tool Family	Tool	Tool Functionality
Environment Management tools	*Startup Launcher* *Main Launcher*	Initialization of the environment, login, launching of other tools, modification of run time parameters
Model Editing Tools	*Diagram Editor*	Manipulation and creation of diagrams where objects and relationships can be viewed and manipulated as graphical diagrams
	Matrix editor	Manipulation and creation of models which can be viewed and edited as matrices, and algorithms performed on them to aid design decisions.
	Table Editor	Manipulation and creation of object types in models and all their properties. Model object types can be viewed together. This is especially useful for requirements analysis.
Model Retrieval tools	*Repository Browsers*	Allows hierarchical access to models and metamodels stored in the repository;
	Report Editor	Generates textual descriptions of the models stored in the repository using a procedural query and data manipulation language.
Method Management Tools	*Object tool* *Property tool* *Relationship tool* *Role tool* *Graph tool*	Specification of conceptual object types and their textual representations
	Symbol Editor	Specification and design of graphical objects and their behaviors. Linking of graphical objects to conceptual object types

Table 5.1. MetaEdit+ Tools

The PARSE-DAT environment has been implemented using the MetaEdit+ metaCASE tool. Although MetaEdit+ supports a multi-user, collaborative editing environment, the particular version we used in building this prototype is a single user version running on Windows NT. However, the model implementation stored in the repository can be migrated to a multi-user (client-server) version running on other platforms such as the various flavours of UNIX.

5.2.2 The GOPRR Meta-Model

The conceptual data model employed by MetaEdit+ is the GOPRR model [MetaCase96d].

The basic GOPRR modeling constructs are:

- A **G**raph denotes an aggregate concept that contains a certain set of objects and their relationships (with specific roles). An example of a graph is a Data Flow Diagram [Yourdon79] [Yourdon89]. The graph concept is fundamentally a generalised decomposition graph: it can be included in a parent graph, attached to an object, role or relationship. Hence, a graph enables modeling and representation of recursive structures such as decomposition, or complex objects as often found in development methods.
- **O**bjects, which consist of independent and identifiable design objects. These typically appear as shapes in diagrams, and can have properties such as names. Objects are basic components of methods. Examples of objects are an Entity in an Entity Relationship Diagram or a Process in a Data Flow Diagram.
- **P**roperties are attributes of objects and can only be accessed as parts of objects or relationships. Properties typically appear as textual labels in diagrams, and they can contain single data entries such as a name, text field or number. An example of a property is the number of a Process in a Data Flow Diagram.
- **R**elationships are associations between objects, and can also have properties. Relationships typically appear as lines between shapes in diagrams, or verbs in texts. An example of a relationship is a Data Flow in a Data Flow Diagram.
- **R**oles define the ways in which objects participate in specific relationships. In diagrams roles typically appear as the end points of Relationships (e.g. an

arrowhead). Roles too can have properties. An example of a role is the specification by directed arrow which end of a data flow relationship is 'to' and which 'from' part of the flow.

Through the use of this GOPRR meta-model, the concepts, languages, graphical representations and operations of a software architecture design method can be supported in MetaEdit+.

5.2.3 Output Specification Tool

There are three environment generators in MetaEdit+.

- The Method support environment generation system compiles the method's object specifications into parts of the metamodel repository when they are defined.
- The Method help generation system generates on-line help components associated with each method. This help can then be accessed through a model editing tool interface from the repository.
- The Report and Transformation generation system. It is used for delivering various reports and conducting checking on the models.

The primary output specification tool is the Report Editor, which is a part of the Report and Transformation generation system. After the conceptual content of the method has been defined, textual representation from the developed models can be produced within MetaEdit+. The textual outputs can be for example check lists, interconnection lists, skeleton program code for the system, or acts as input to external model analysis tools. Such outputs can be defined within MetaEdit+ using the Report Editor tool.

The Report Editor is based on the use of templates. The templates provide access to the programming constructs in the Report Definition Language. The Report Definition Language is similar to a 3GL providing sequence, selection and iteration, as well as references to each element in the GOPRR model, i.e. Graphs, Objects, Properties, Roles and Relationships.

Once the report has been specified, it can be tested by selecting the *Run* menu option. The reports are automatically tied to the method used, and thus are loaded automatically when the method is selected.

5.3 PARSE-DAT Design Environment (PARSE-DT)

5.3.1 Implementation

The development of PARSE-DT involved the following steps:

1. Development of the method data-model
2. Development of representations
3. Evaluation of the models
4. Development of reports and model checking
5. Generation of support environment and guidance

The steps were carried out partially in parallel, and there were iterations through the steps. The evaluation of the model led to redesign and corrections, and the use of the method led to its subsequent evolution.

The primary focus of the development of the Dynamic PARSE data model involves an examination of the conceptual content of the Dynamic PARSE method, i.e. identifying the objects and their relationships in the method. Once these objects have been identified, they are visually represented using the GOPRR notation [Metacase96d]. The conceptual structure forms the core of the PARSE-DT tool development, because it defines the data-model of the PARSE-DT environment, and is used as the source of all other definitions and tool functionality. Figure 5.2 shows the GOPRR model of Dynamic PARSE.

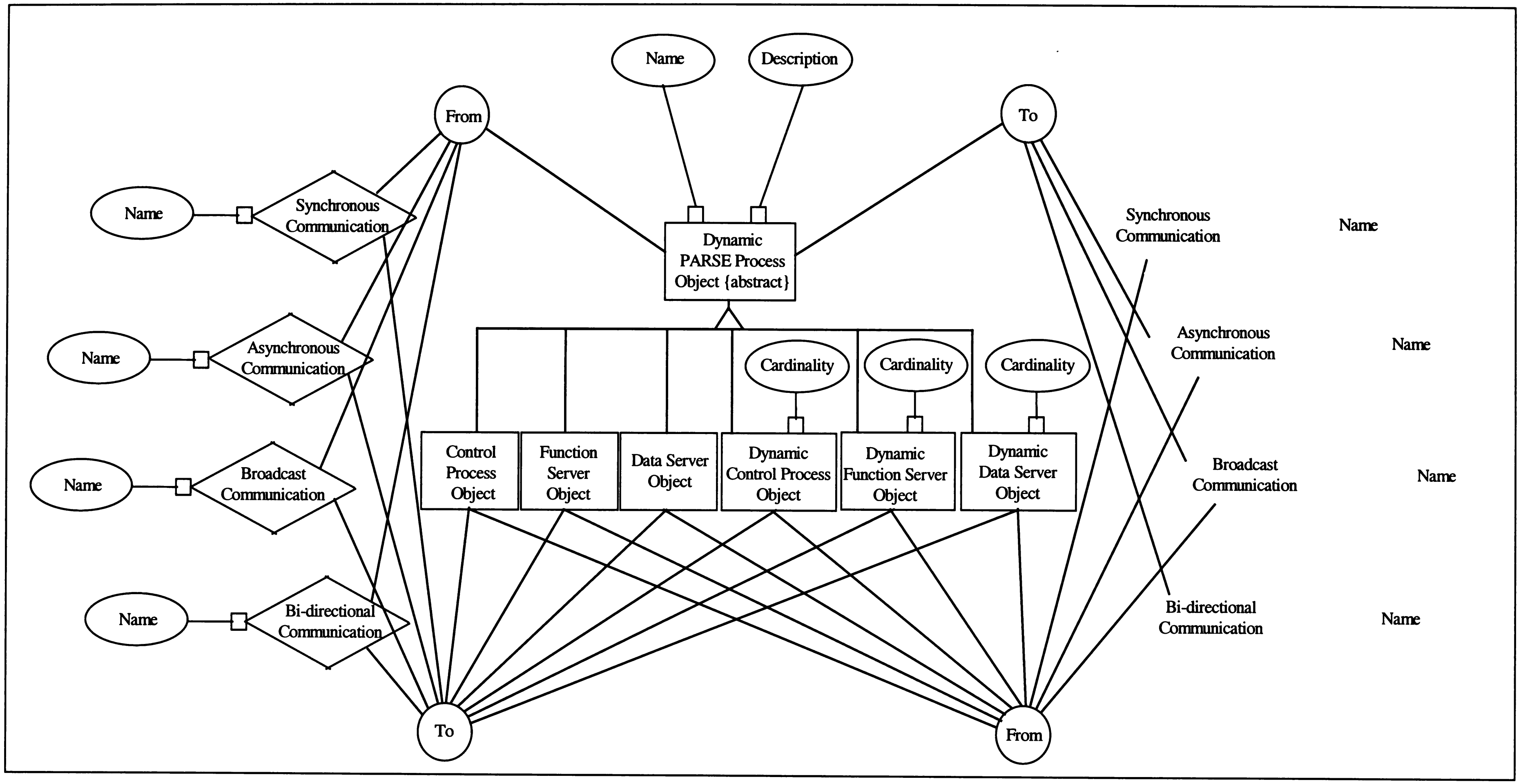

Name
Description
From
To
Dynamic PARSE Process Object {abstract}
Name
Synchronous Communication
Name
Asynchronous Communication
Name
Broadcast Communication
Name
Bi-directional Communication
Cardinality
Cardinality
Cardinality
Control Process Object
Function Server Object
Data Server Object
Dynamic Control Process Object
Dynamic Function Server Object
Dynamic Data Server Object
Synchronous Communication
Name
Asynchronous Communication
Name
Broadcast Communication
Name
Bi-directional Communication
Name
To
From

Figure 5.2 GOPRR Model of Dynamic PARSE

The similarities between the various types of Dynamic PARSE Process Objects led to the abstract parent object. These similarities include the properties of name and description, as well as the connections via the various types of communication paths. This inheritance hierarchy can be seen in the centre of Figure 5.2, where the rectangles are object types, diamonds are relationship types, circle role types, and ovals property types. Dynamic PARSE Process Objects thus has two properties: Name and Description, and the three Dynamic Process Objects inherit these properties, as well as adding another: Cardinality. Process objects are connected by communication paths via To and From roles.

5.3.2 Using PARSE-DT

Dynamic PARSE designs are constructed in the PARSE-DAT design environment (PARSE-DT). Figure 5.3 shows the main graph editor window, where Dynamic PARSE designs can be constructed using various Dynamic PARSE design components provided in the palette. This supports rapid prototyping. The main MetaEdit+ control window is also displayed, from which the graph editor window can be invoked.

The software architecture design of an example *Data_Control* module has been constructed using various tools in the palette, and is displayed in the main graph editing window. There are two passive data server process objects, coordinated by two control process objects. The four process objects are connected by various communication paths.

Design rules are built into the PARSE-DT environment, hence imposing restrictions on designs. For example, if a designer attempts to connect an asynchronous communication path to a data process object, the editor will display a message window indicating that this is not allowed, as it is meaningless to send messages to a data server asynchronously. This feature removes many common design errors.

Designers can also specify various properties about the individual design components. Such properties include for process objects - process object name, behavioural description, decomposition (for composite process objects), and for communication paths: path name, type of message, protocol and priority.

The PARSE-DAT tool may be used in a multi-user mode, due to the support from MetaEdit+. Thus, collaborative software architecture design work can be carried out amongst a design team, and the precise usage of the Dynamic PARSE design notation eliminates any possibility of design ambiguity or confusion.

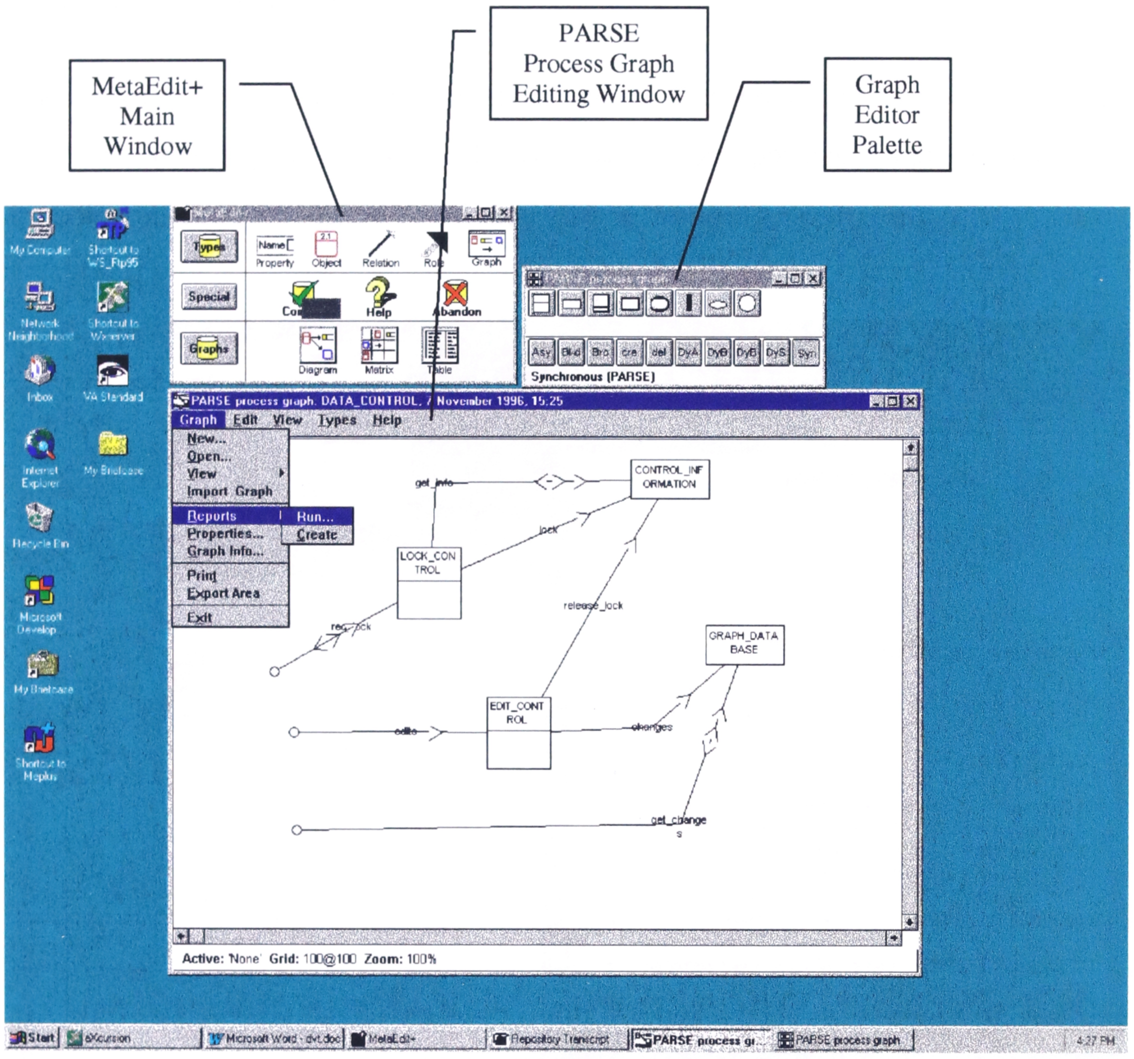
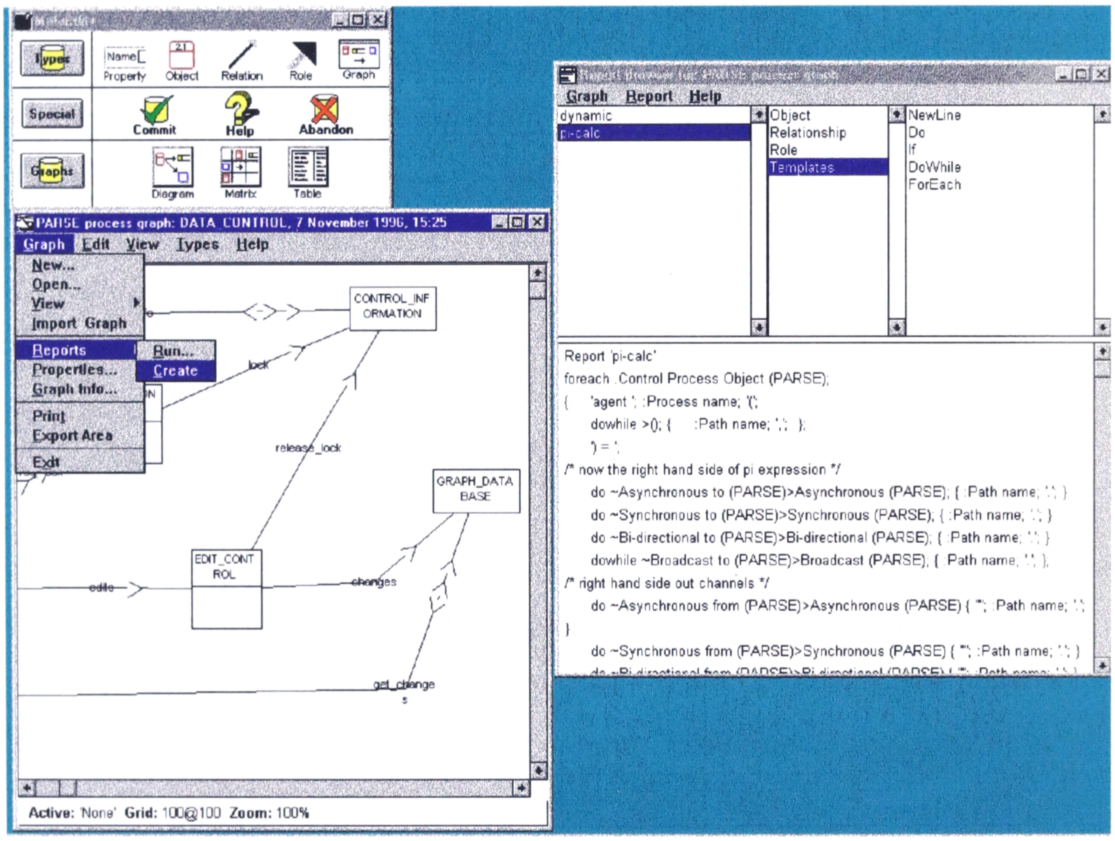


Figure 5.3 PARSE-DT Screen Capture

5.4 The Design Analysis Environment (PARSE-AT)

5.4.1 Implementation

The design analysis environment consists of a model transformation processor, built using the Meta-Edit+ Report Generator. This generates a corresponding π-Calculus model in textual form. These π-Calculus models may then be analysed for correctness.



```
Report 'pi-calc'
foreach .Control Process Object (PARSE);
{     'agent '; :Process name; '(';
      dowhile >(); {     :Path name; ',';  };
      ') = ';
/* now the right hand side of pi expression */
      do ~Asynchronous to (PARSE)>Asynchronous (PARSE); { :Path name; '.'; }
      do ~Synchronous to (PARSE)>Synchronous (PARSE); { :Path name; '.'; }
      do ~Bi-directional to (PARSE)>Bi-directional (PARSE); { :Path name; '.'; }
      dowhile ~Broadcast to (PARSE)>Broadcast (PARSE); { :Path name; '.'; };
/* right hand side out channels */
      do ~Asynchronous from (PARSE)>Asynchronous (PARSE) { '"'; :Path name; '.';
}
      do ~Synchronous from (PARSE)>Synchronous (PARSE) { '"'; :Path name; '.'; }
      do ~Bi-directional from (PARSE)>Bi-directional (PARSE) { '"'; :Path name; '.'; }
```

Figure 5.4 π-Calculus Report Generator

The control structures available for use include Do, DoWhile and ForEach (iterations), and If (selections). The iterations can be applied to all of the GOPPR data model entities, which correspond to the PARSE-D process graph, the various process objects, the communication paths, as well as their respective properties and roles. In the translation, the process object names' have been used as agent names, and the communication path names used as πcomposition of all the static process objects. Figure 5.4 shows the Report generator in PARSE-DAT.

5.4.2 External Design Analysis Tool: Mobility Workbench

Once a Dynamic PARSE design has been constructed in the PARSE-DT environment, the corresponding π-Calculus can be automatically generated for analysis/verification purpose. The π-Calculus generator has been implemented by utilising the MetaEdit+ Report Editor.

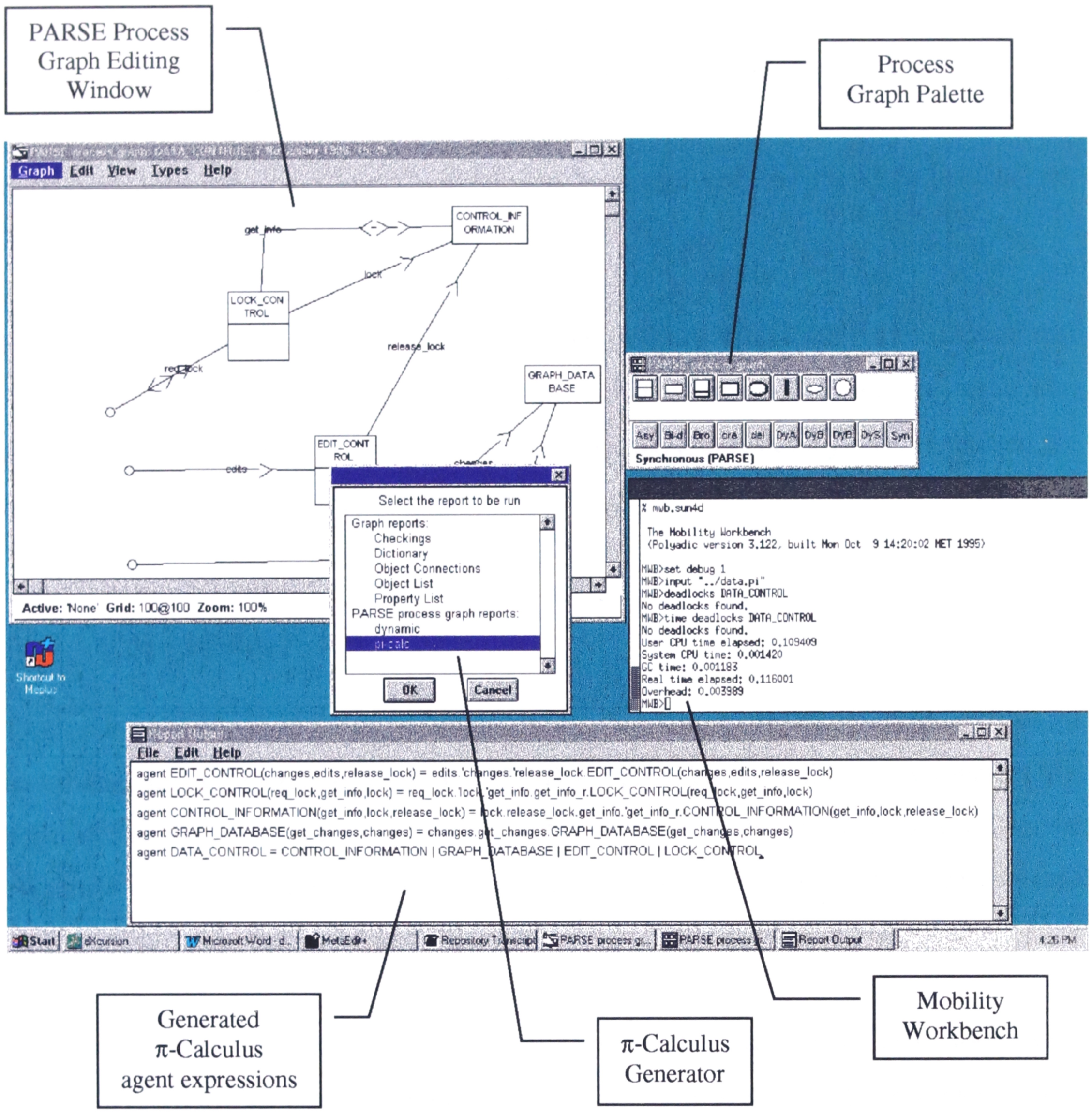
The Mobility Workbench is a model checker for π-Calculus. It has the following capabilities:

- syntactical checks on π-Calculus expressions
- equivalence checks on multiple π-Calculus expressions
- deadlock detection (with output trace)
- deadlock freedom verification
- textual file input
- timing information
- debugging messages

The Mobility Workbench employs the *on-the-fly* search algorithm to alleviate the state explosion problem. The On-the-fly algorithm relies on demand-driven generation of states to avoid the construction of irrelevant system configurations [Andersen94] [Bhat95].

5.4.3 Using PARSE-AT

Figure 5.4 shows how users can invoke the π-Calculus report generator in the PARSE design analysis and verification environment (PARSE-AT).



```
% mwb.sun4d

 The Mobility Workbench
 (Polyadic version 3.122, built Mon Oct  9 14:20:02 MET 1995)

MWB>set debug 1
MWB>input "../data.pi"
MWB>deadlocks DATA_CONTROL
No deadlocks found.
MWB>time deadlocks DATA_CONTROL
No deadlocks found.
User CPU time elapsed: 0.109409
System CPU time: 0.001420
GC time: 0.001183
Real time elapsed: 0.116001
Overhead: 0.003989
MWB>
```

```
agent EDIT_CONTROL(changes,edits,release_lock) = edits.'changes.'release_lock.EDIT_CONTROL(changes,edits,release_lock)
agent LOCK_CONTROL(req_lock,get_info,lock) = req_lock.'lock.'get_info.get_info_r.LOCK_CONTROL(req_lock,get_info,lock)
agent CONTROL_INFORMATION(get_info,lock,release_lock) = lock.release_lock.get_info.'get_info_r.CONTROL_INFORMATION(get_info,lock,release_lock)
agent GRAPH_DATABASE(get_changes,changes) = changes.get_changes.GRAPH_DATABASE(get_changes,changes)
agent DATA_CONTROL = CONTROL_INFORMATION | GRAPH_DATABASE | EDIT_CONTROL | LOCK_CONTROL
```

Figure 5.5 PARSE-AT Screen Capture

The user can firstly generate the corresponding π-Calculus expressions from the graphical design by invoking the π-Calculus generator. The resultant π-Calculus agent expressions are displayed in the 'Report Output' window. This textual output can then act as textual file input to the Mobility Workbench [Victor94], where analysis and verification can be carried out. Currently, the Mobility Workbench runs on Windows NT (as its corresponding license on UNIX is around 10 times more expensive), and the University facility provides more powerful Sun machines which are suited for running the mobility Workbench. Hence, we have adopted the textual file sharing method for integrating the use of Mobility use of Mobility Workbench's Report Editor and the Mobility Workbench. In future versions, this mechanism can be improved through either of two means:

- purchase MetaEdit+ license for UNIX platforms, or
- recompile Mobility Workbench to produce executables running on PCs.

In this example, the *Data_Control* module has been analysed and shown to be deadlock free.

In some other cases, the analysis result from the Mobility Workbench may indicate that the design is not deadlock free. In which case, that information should be taken into account in the next refinement phase. Alternatively, the architect may wish to discard the faulty design altogether and opt for a different design.

5.5 Conclusion

This chapter has presented an environment for supporting the design and analysis of distributed software architectures. PARSE-DAT provides an editing environment for software architects to specify software architectures quickly and precisely using the Dynamic PARSE Process Graph Notation. The resultant designs can then be analysed and checked for structural deadlocks.

The PARSE-DAT environment has been implemented using a metaCASE tool called MetaEdit+. The advantage of taking this approach is twofold: by using a meta-CASE tool, which is a specialised application generation environments, less tool development time and resource is required compared to implementation using 3GLs. At the same time, the resultant tool/method can be fully customised according to the methodology, and is more powerful and useful compared to using general-purpose applications.

The various features of the PARSE-DAT environment include the following:

- Easy to use graphical editing environment (PARSE-DT), which includes completeness and consistency checks.
- Automatic transformation of graphical design to corresponding formal model (PARSE-AT).
- Support for external design analysis (PARSE-AT).

The Dynamic PARSE software design methodology coupled with the supporting environment thus encourages an iterative design process by enabling software architects to check for design correctness of various alternate designs at an early stage of engineering process.

The advantages of this early iterative design process include:

- Early detection of design fault reduces cost in development.
- Iterative verification of alternate design choices allows software architects to choose the better design.

6. Case Studies

This chapter presents four case studies designed and analysed using the Dynamic PARSE methodology.

The case studies range from the simple client-server and pipeline architectural style, to a novel design of a high speed network, and a collaborative work environment.

For each of these case studies, the Dynamic PARSE architectural design will be presented. This is followed by the corresponding design analysis and verification carried out using PARSE-DAT.

6.1 A Client-Server System

6.1.1 Dynamic PARSE Design

Client-server computing exploits all the strong points of a distributed system, such as fault tolerance and application portability. Developing software using the client-server paradigm encourages code reuse. In a networking environment, a server often serves multiple clients [Comerford94] [Larocque94]. In order to exploit the hardware parallelism, the *main server* often creates multiple *workers* to provide services to all clients.

In the top-level diagram (Figure 6.1), *client* is modeled to be a dynamic process. This means there may be multiple clients at any one time, or there may be no clients at all. An external object called *client_generator* is used to create clients in the system. In real world applications, the client request may come from the user keying a request through the keyboard, or may simply be a part of an application program needing certain computations carried out by another program.

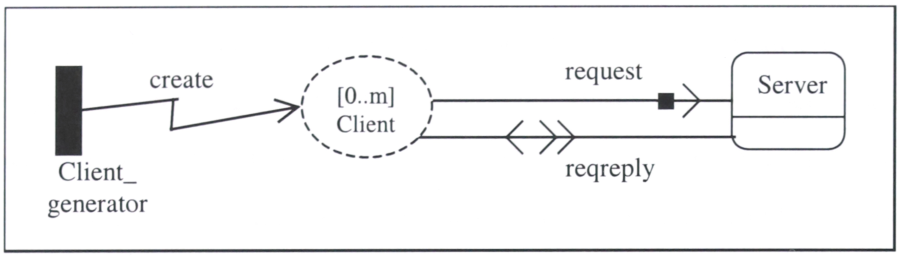


Figure 6.1. Top Level Process Graph of a Client-Server System

There are two communication paths between a *client* and the *server*. There is a non-blocking send from the *client* to *request* service to be done, with a blocking receive at the *server's* end. Then, a bi-directional communication path is set up, for the actual *request* message and *reply* to be transferred.

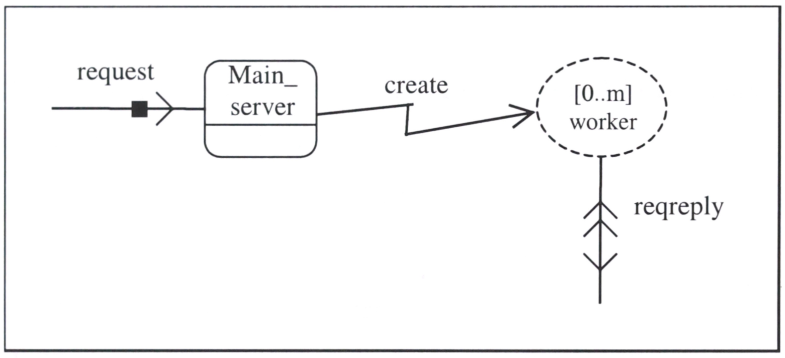


Figure 6.2. Decomposition of the server process

The *server* process here is modeled as a control process. Its decomposed structure is as shown in Figure 6.2. Its function is to receive requests from clients, create instances of *workers* to perform the necessary work or computation, and upon completion, send the required result back to *clients*. Notice that the work *request* and *reply* messages are passed between *clients* and the *worker* processes without the supervision of the *main_server* process. Upon work completion, instances of the *worker* process objects exit the system. Hence the default termination behaviour: aging.

6.1.2 Dynamic PARSE Analysis and Verification

The following is the corresponding partial π-Calculus model generated by PARSE-AT for the purpose of checking for structural deadlocks. Basically, the process objects have been translated into π-Calculus agents with possible actions or events being possible messages passed along the various communication paths, which essentially represent the dependency relationships between the various process objects in the system. Notice reliable communication is assumed, hence the simplified translation for asynchronous communication paths. This simplification does not alter the deadlock checking result. The creation of *WORKER* process is similarly simplified. In addition to achieving a π-Calculus model that is easier to understand (for the presentation purpose in this thesis), this simplification also reduces the search space, hence reducing the time required for the analysis process.

agent CLIENT(reqreply,request) = 'request.'reqreply.CLIENT<reqreply,request>
agent CLIENTSERVER = (SERVER | CLIENT)
agent SERVER = (MAINSERVER | WORKER)
agent MAINSERVER(create,request) = request.'create.MAINSERVER<create,request>
agent WORKER (reqreply,create) = create.reqreply.WORKER<reqreply,create>

The overall system is thus a parallel composition of all the process objects, namely the *CLIENTSERVER* agent. A deadlock check can then be carried out to detect any structural deadlock in this client-server software architecture. Also note that the composite *SERVER* object can firstly be analysed for deadlocks, as this π-Calculus model supports compositional analysis for a hierarchical Dynamic PARSE design. The following is the analysis output from Mobility Workbench:

```
MWB>deadlocks CLIENTSERVER
No deadlocks found.
MWB>deadlocks SERVER
No deadlocks found.
```

The analysis results from Mobility Workbench indicate that this client-server architecture is free from structural deadlocks. Hence, the designer have gained some confidence in this design by using PARSE-DAT, and may continue to refine the design, and or further develop the system using for example an OOD method [Eriksson98].

6.2 Primes Sieve of Erastosthenes

6.2.1 Dynamic PARSE Design

This example is the classic problem "Primes Sieve of Erastosthenes". This problem of generating prime numbers using multiple processes is described in [Kramer85]. The aim is to compute all the prime numbers up to a certain positive integer *limit*. To achieve this, we require a process that generates a stream of odd numbers, a set of processes that filters out non-primes, and an output process that returns prime numbers. However, we do not know in advance how many *filtering* processes are required since we do not even know how many prime numbers to compute. Dynamic PARSE allows the design of this type of program, where the structure cannot be determined statically.

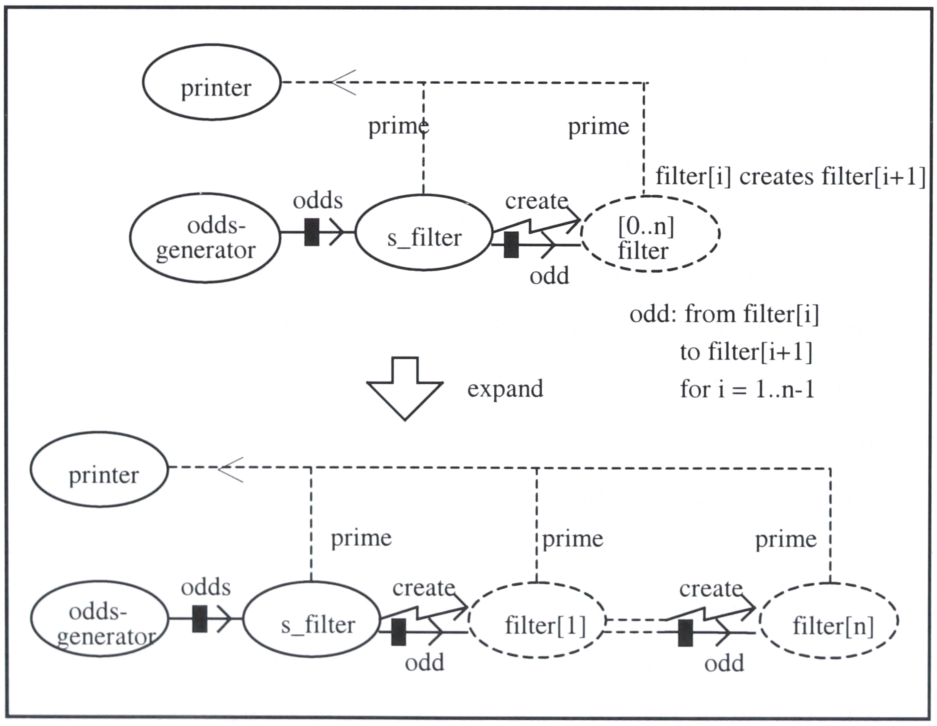


Figure 6.3. Dynamic PARSE Design of 'Primes Sieve of Erastosthenes'

The process graph in Figure 6.3 computes prime numbers. The process *odds_generator* generates a stream of odd numbers that are fed into a pipeline of processes. Each *filter* process sends the first number it receives to a *printer* process and subsequently filters out multiples of that number from the stream of odd numbers. It also creates a new *filter* process when it detects a new prime. The program terminates when a filter sends a prime to the *printer* process that is greater than limit.

For example, *odds_generator* generates 3 as the first number, and passes it to the first *filter* process. This *filter* process passes 3 as a prime number to the *printer* process, and the *printer* process subsequently prints 3. The next number generated is 5, since 5 is not a multiple of 3, the first *filter* process creates a second *filter* process, and passes 5 to it. Upon receiving the number 5, the second *filter* process sends 5 to the *printer* process for printing. This process is continued for number 7, hence a third *filter* process is created. When 9 has been generated, it however does not bypass the first *filter* process. Since 9 is a multiple of 3, it is discarded. We can see from this process, a sequence of prime numbers is generated: 3, 5, 7, 11, 13, ... and so on, with each prime number received by the last *filter* in the pipeline at all times.

The following is a set of design features:

- The *s_filter* process is static. It is created when the system begins execution.
- Value of n depends on limit. It is determined at run time. This is consistent with *filter* processes being created at run time.
- The creation of *filter* processes in the *growing* pipeline is ordered. Each *filter* process is created by the last created process. The unique numbering of *filter* process instances can be used to define the ordering here: *filter[2]* is created after *filter[1]*, *filter[3]* is created after *filter[2]*, and so on. Hence, for example, *filter[4]* does not exist in the system if *filter[i] for i = 1,2, and 3* are not present in the system.
- Once an instance of the *filter* process has been created, it stays in the system until the termination of the entire program itself. That is, when all prime numbers up to *limit* have been computed.
- The paths named *prime* going into the printer process are *dotted*. There is always just one prime number coming from the last filter in the pipeline. As soon as a particular filter instance has used its *prime* path to send a number to *printer*, that path is no longer needed. Hence the transactional nature of this communication path.

- The printer process may be modeled as a control process if it controls the termination of the program. Whenever the printer process receives a prime number from an instance of the filter process, it checks to see if it is greater than *limit*. If it is, then the program is terminated.
- The pipeline may grow forever. If *limit* is specified as *unlimited* or not specified at all, then the printer process accepts all numbers coming from the *prime* communication path, and new *filter* processes are generated all the time (although becoming less frequent!).

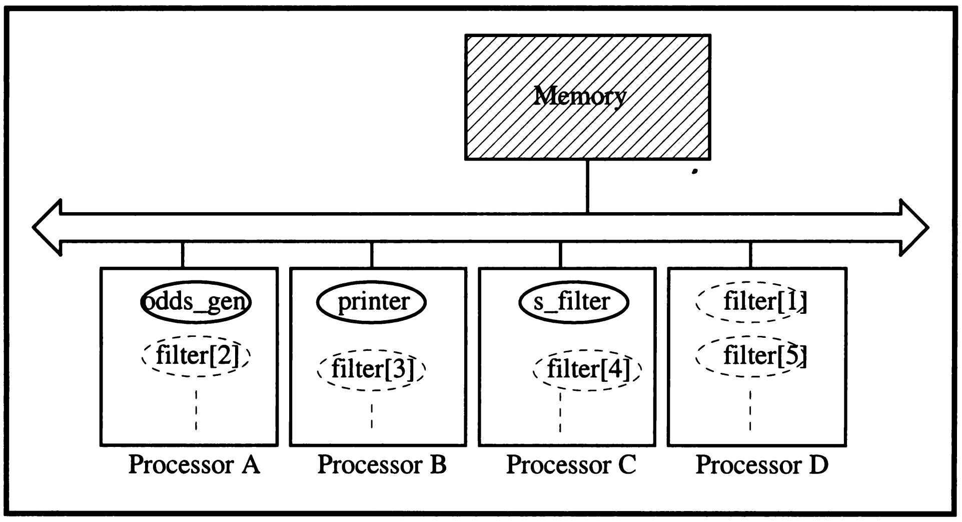


Figure 6.4. Process Distribution on a Multi-Processor Machine

The solution to this problem is a perfect example of an application program that can fully exploit the processing power of symmetric multi-processing machines. Each time a *filter* process is created, the underlying system would dispatch it to a processor with the smallest load. So, basically, all *filter* processes could be more or less evenly distributed around different processors.

6.2.2 Dynamic PARSE Analysis and Verification

The corresponding π-calculus model to the Sieve of Erastosthenes design is as follows:

agent ODD_GENERATOR(odd,x) = $\overline{x}$ *<odd>.ODD_GENERATOR(odd,x)*

agent PRINTER(y) = y(prime).PRINTER(y)

agent FILTER(x,y) = x(odd). $\overline{y}$ *<odd>.FILTER'(x,y)*

agent FILTER'(x,y) = x(odd).(FILTER'(x,y) + (^mid)(FILTER''(x,mid) | FILTER(mid,y)))

agent FILTER''(x,mid) = x(odd). $\overline{mid}$ *<odd>.FILTER''(x,mid)*

The names *x* and *y* have been introduced to model the incoming and outgoing path respectively for passing the *odd* number from one *filter* to the next. The name *mid* has been introduced to model the intermediate connection between two adjacent *filter* processes. Notice that the name *odd* represents the data (i.e. an odd number) being passed through the pipeline of *filter* processes. Hence, this model is data dependent, and is a more complete model of the Dynamic PARSE design.

However, this π-calculus model is infinite in nature. It generates an infinite search space, and hence any analysis on structural deadlocks would be inconclusive. However, one should notice that the behaviour of all the *filter* processes are identical. Hence, the π-calculus model used for deadlock checking can be reduced to the following:

agent ODD_GENERATOR(odds) = 'odds.ODD_GENERATOR(odds)

agent PRINTER(prime) = prime.PRINTER(prime)

agent SFILTER(odds,odd,prime) = odds.'prime.'odd.SFILTER'(odds,odd)

agent SFILTER'(odds,odd) = odds.'odd.SFILTER'(odds,odd)

agent FILTER(odds,odd,prime) = odds.'prime.'odd.FILTER'(odds,odd)

agent FILTER'(odds,odd) = odds.'odd.FILTER'(odds,odd)

agent SIEVE = SFILTER | FILTER | ODD_GENERATOR

This PARSE-DAT generated π-calculus model can then be analysed by Mobility Workbench, and has been found to be free from structural deadlocks.

6.3 HTPNET

6.3.1 Dynamic PARSE Design

This network transfer protocol example is taken from [Gorton94], which is an experience report on the design and implementation of the packet management component of a transport protocol for broadband networks. The protocol, known as HTPNET [Chan94], has been designed to exploit parallel architectures. It consists of the protocol software at both the transmitter node and the receiver node, as well as a simulation of a high speed network to connect transmitter and receiver. In this case study, only the design of the software for the receiving node will be considered in detail.

The highest level of the system design is shown in Figure 6.5. *Transmit* and *Receive* respectively encapsulate the required behaviour of sending and receiving packets. Internally, each is decomposed into a number of lower level process objects that perform the protocol processing. Both are active process objects with complex internal state, and hence are most appropriately represented in a process graph by the control process icon.

Network is categorised as a function server. It is passive, sequential, has no externally visible state and encapsulates some well-defined functionality within the system. It simply accepts data and synchronisation packets from *Transmit*, and passes these on to *Receive* after introducing a realistic delay. In a similar manner it relays acknowledge packets from *Receive* to *Transmit*.

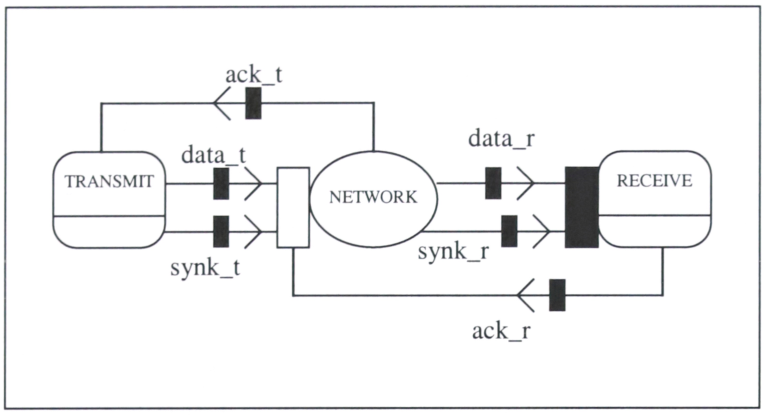


Figure 6.5. Top Level PARSE Design

As there is no inherent prioritisation amongst its input paths, *Network* selects data non-deterministically from one of the three input paths. This is contrasted against *Receive*, which handles its two input paths concurrently. *Receive* is required to use different process objects to accept data packets from each path at a lower level abstraction. This implements the semantics of a concurrent path constructor, and is a design rule that can be checked and enforced.

The decomposition of *Receive* is illustrated in Figure 6.6. *Receive* consists internally of three process objects. *DataProc* is responsible for processing data packets from *Network*, and *SynkProc* is responsible for processing incoming synchronisation packets and generating acknowledgments. This satisfies the requirements placed on the decomposition of *Receive* by the concurrent path constructor in the top level design diagram. *DataProc* is modelled as a control process because, in addition to processing data packet headers, it maintains state information on successfully received data packets. This state information is periodically retrieved by *SynkProc* over the *Status* path, which provides a synchronised request-reply connection. *SynkProc* uses the status information to produce an output on the *Ack* path which is relayed via *Network* to *Transmit*. *DataProc* imposes no priority over its two input paths, indicated by their convergence into a non-deterministic path constructor.

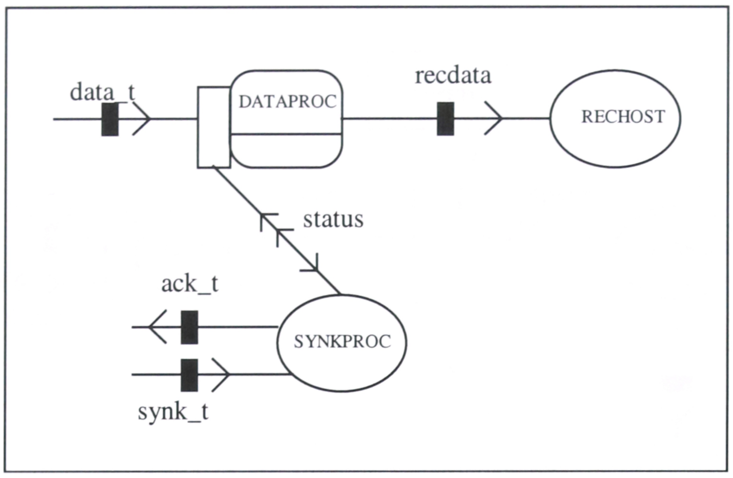


Figure 6.6. RECEIVE Object Decomposition

When a packet is found to be error-free and in-sequence, *DataProc* sends the corresponding packet address reference to the *RecHost* process object. *RecHost* represents the operating system interface on the receiving node. It essentially acts as a packet sink, extracting messages from the *RecData* asynchronous communications path.

Moving down to the lowest level of decomposition for *Receive*, the internal structure of *DataProc* is shown in Figure 6.7. The *Header* function server non-deterministically accepts messages on the *Data* and *Status* paths. Packet headers are stored in the *tempFIFO* data server process. A synchronous path connects Header to *tempFIFO*, thus requiring the two processes to rendezvous to exchange data. Synchronous paths are generally most suitable for connecting to data servers. Passing data over a buffered asynchronous path to a data server which already provides buffering introduces the potential for unnecessary copying of data.

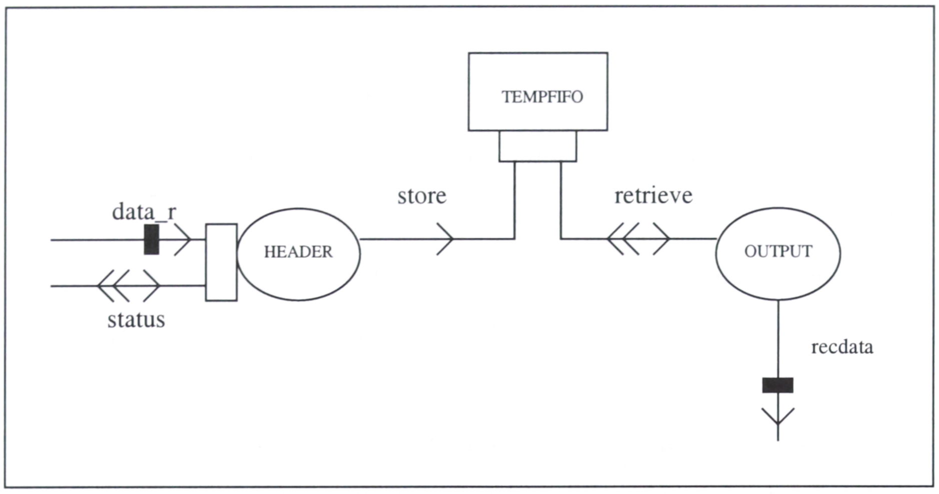


Figure 6.7. DATAPROC Object Decomposition

The process object *tempFIFO* is a data server that simply waits on its two input paths and selects one non-deterministically. It essentially provides some post-processing (header stripping) and a FIFO buffer between the *Header* and *Output* function servers.

6.3.2 Dynamic PARSE Analysis and Verification

The corresponding π-calculus model for this partial HTPNET system is as follows:

$agent\ HEADER(data\_r, status, store, st) =$

$\qquad data\_r(d).\,\overline{store}<d>.HEADER(data\_r, status, store, st)$

$\qquad +\ status(sr).\,\overline{sr}<st>.HEADER(data\_r, status, store, st)$

$agent\ TEMPFIFO(store, retrieve) = store(d).TEMPFIFO(store, retrieve)$

$\qquad +\ retrieve(req).\,\overline{req}<d>.TEMPFIFO(store, retrieve)$

$agent\ OUTPUT(retrieve, req, recdata) = \overline{retrieve}<req>.req(d).\,\overline{recdata}<d>$

$\qquad .OUTPUT(retrieve, req, recdata)$

$agent\ DATAPROC = HEADER \mid TEMPFIFO \mid OUTPUT$

$agent\ SYNKPROC(synk\_r, status, ack\_r, sr) = synk\_r(s).\,\overline{status}<sr>.sr(st).\,\overline{ack\_r}<st>$

$\qquad .SYNKPROC(synk\_r, status, ack\_r, sr)$

$agent\ RECHOST(recdata) = recdata(rd).RECHOST(recdata)$

$agent\ RECEIVE = DATAPROC \mid SYNKPROC \mid RECHOST$

$agent\ TRANSMIT(data\_t, synk\_t, ack\_t, d, s) =$

$\qquad \overline{data\_t}<d>.TRANSMIT(data\_t, synk\_t, ack\_t, d, s)$

$+ \overline{synk\_t} <s>.ack\_t(ac).TRANSMIT(data\_t,synk\_t,ack\_t,d,s)$

agent $NETWORK(data\_t,ack\_t,synk\_t,data\_r,ack\_r,synk\_r,sr) =$

$data\_t(d).\overline{data\_t} <d>.NETWORK(data\_t,ack\_t,synk\_t,data\_r,ack\_r,synk\_r,sr)$

$+ synk\_t(s).ack\_t(ac).NETWORK(data\_t,ack\_t,synk\_t,data\_r,ack\_r,synk\_r,sr)$

$+ \overline{synk\_r} <sr>.ack\_r(ac).NETWORK(data\_t,ack\_t,synk\_t,data\_r,ack\_r,synk\_r,sr)$

agent $SYSTEM = TRANSMIT | NETWORK | RECEIVE$

The names such as *d* and *st* are data being passed along communication paths. They have been included in this model to improve readability. They represent the data being passed around and the status information. The π-calculus model generated by PARSE-DAT does not contain these names in order to reduce search space. This simplication does not affect the deadlock analysis. Further, the asynchronous communication channels have been simplified to be synchronous communication channels in the corresponding π-calculus specification, for both presentation and analysis efficiency purposes. Also, the hierarchical structuring of design is reflected in the π-calculus model.

Upon analysing these π-calculus models for structural deadlocks, the following output was obtained from Mobility Workbench:

```
MWB>deadlocks TRANSMIT
No deadlocks found.
MWB>deadlocks NETWORK
No deadlocks found.
MWB>deadlocks RECEIVE
No deadlocks found.
MWB>deadlocks SYSTEM
No deadlocks found.
```

Hence, we find there is no structural deadlocks present in this system. Notice that it is possible to expand the π-calculus model for the *NETWORK* and *TRANSMIT* module if their graphical design is given. Once again, the advantage of compositional analysis is demonstrated here.

Appendix A lists the π-calculus model and analysis result of an intermediate HTPNET design which exhibits the presence of structural deadlock.

HTPNET has been implemented in Occam running on a network of transputers. For the implementation details of the HTPNET system (including process graphs and nets for the *Transmit* and *Network* process objects), please refer to [Chan93].

Lastly, this example design illustrates that the analysis performed by the Mobility Workbench tool is also applicable to static process architectures.

6.4 SEP-TOOL

6.4.1 Dynamic PARSE Design

This case study is based on a research project described in [Gorton97] and [Hawryszkiewycz96] which looks at the application of computer-supported collaborative work (CSCW) or groupware technology to a range of software engineering activities. Groupware technology provides support for collaboration on software development tasks, and is applicable to both co-located and geographically distributed software development teams.

The basic goals of this GWSE (Global Working in Software Engineering) system were to provide the following:

- software process description tools
- project management and monitoring facilities
- workflow management for defined processes
- transparent document sharing, archiving and management

Since the requirement was to support both co-located and widely distributed software development teams, the system must be able to provide distributed coordination, communication and document management between groups of developers at several sites.

Several commercial-off-the-shelf (COTS) components have been integrated, and supplemented with additional functionality to form the GWSE system. The workflow management engine is based on Lotus Notes, the document archive database is based on Intersolv's PVCS version control system, the process modeling tool SEP-Tool, has been implemented in C++ using CORBA as a distribution and synchronisation technology, and the modeling tool interfaces with the project management tools and capabilities which are provided by Microsoft Project.

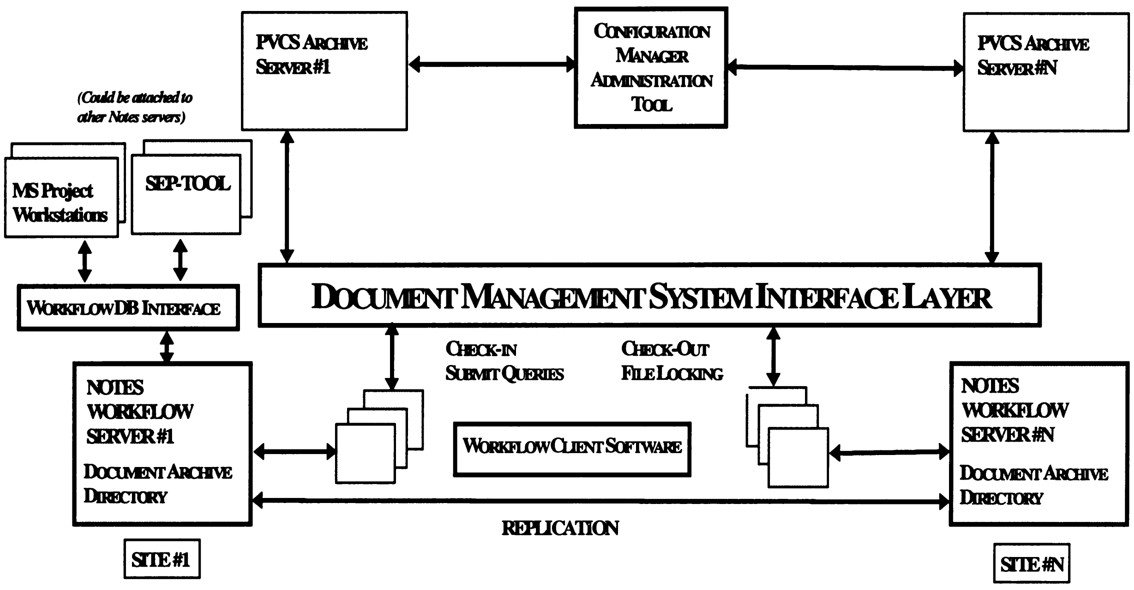

PVCS Archive Server#1
Configuration Manager Administration Tool
PVCS Archive Server#N
(Could be attached to other Notes servers)
MS Project Workstations
SEP-TOOL
WorkflowDB Interface
Document Management System Interface Layer
Check-In Submit Queries
Check-Out File Locking
Notes Workflow Server#1 Document Archive Directory
Workflow Client Software
Notes Workflow Server#N Document Archive Directory
Replication
Site#1
Site#N

Figure 6.8. GWSE Architecture [Gorton97]

This case study focuses on the software process modeling as provided by the SEP-TOOL (Software Engineering Process-TOOL). This is a graphical process modeling environment which enables distributed team members to collaboratively define a process model for a project. The tool is a real-time groupware tool, or WYSIWIS (What You See Is What I See), and needs to ensure model consistency when faced with multiple concurrent updates.

Applications such as the SEP-TOOL have many complex concurrency requirements. For example, the client must be able to accept updates and various messages while the graphical user interface is in use. And at the server side, updates to the consistent model must be broadcast to all clients, while accepting new requests. Further, new clients need to be initialised while broadcasting outstanding updates. Lastly, an additional requirement is the support for editing multiple models simultaneously.

There are many different server implementation possibilities. These requirements naturally lend themselves to multithreaded solutions, such as: thread-per-update-request, thread-per-client, thread-per-update-broadcast, and so on.

In order to minimise risks and cost of failure, an iterative approach should be taken such that complex components in the architecture are isolated and demonstrated by building 'proof of concept' prototypes.

PARSE-DAT can be used to analyse the critical components of the architecture. Figure 6.9 shows an architectural component of the SEP-Tool that has been analysed using PARSE-DAT.

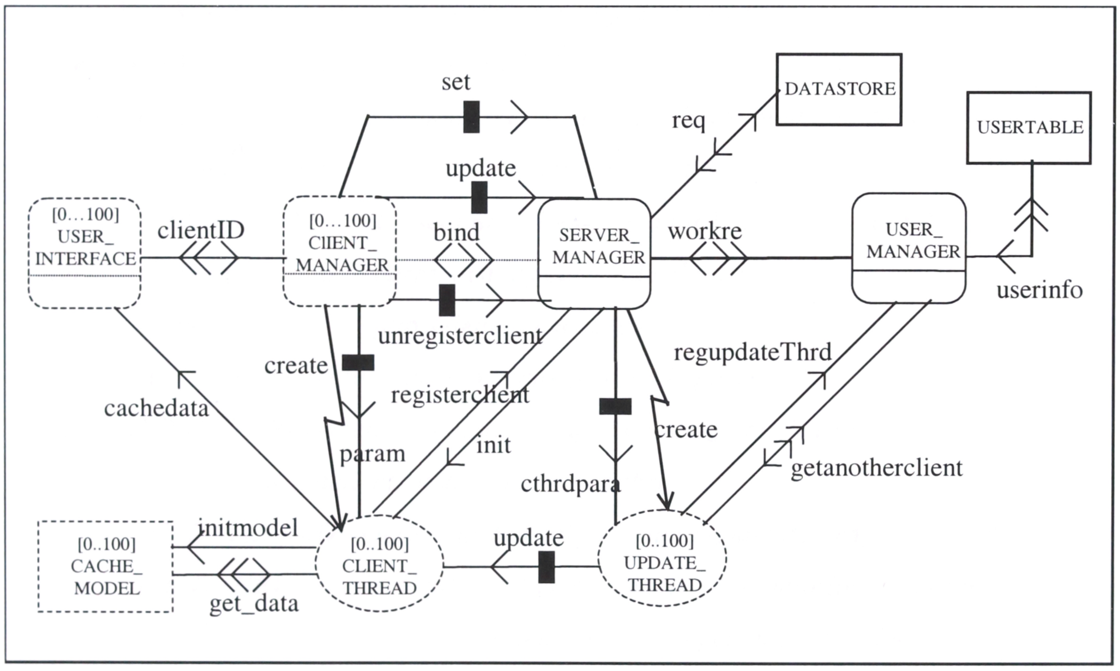


Figure 6.9. Dynamic PARSE Design for SEP-Tool

6.4.2 Dynamic PARSE Analysis and Verification

The following is the generated π-Calculus model for the SEP-TOOL architecture in Figure 6.9. Notice the peripheral objects such as *datastore* and *usertable* are not captured in this π-Calculus model. Since these passive data servers communicate only with the *server* and *usermanager* process objects, their dependency relationship does not need to be analysed for deadlock. The advantage of this is that the search space is reduced.

```
agent SERVER(regclient,set,unregclient,bind,cthrdparam,init,workreq,req,update) =
        regclient.set.unregclient.update.init.'cthrdparam.'workreq.'req
        .SERVER(regclient,set,unregclient,bind,cthrdparam,init,workreq,req,update)
agent USERMAN(getclient,workreq,userinfo,regupdatethrd) =
        getclient.regupdatethrd.workreq.'userinfo
        .USERMAN(getclient,workreq,userinfo,regupdatethrd)
agent USERTABLE(userinfo) = userinfo.USERTABLE(userinfo)
agent DATASTORE(req) = req.DATASTORE(req)
agent CLIENTMAN(clientid,set,unregclient,cclienthrdpara,bind,update) =
        'set.'unregclient.'cclienthrdpara.'update.'clientid
        .CLIENTMAN(clientid,set,unregclient,cclienthrdpara,bind,update)
agent UI(clientid,clientinfo,cachtodisplay,cache4display) =
        clientinfo.cachtodisplay.clientid.'cache4display
        .UI(clientid,clientinfo,cachtodisplay,cache4display)
agent UPDATETHRD(getclient,update,cthrdparam,regupdatethrd) =
        cthrdparam.'getclient.'update.'regupdatethrd
        .UPDATETHRD(getclient,update,cthrdparam,regupdatethrd)
agent CLIENTTHRD(regclient,initmodel,getcacheddata,cclienthrdpara,update,
        cachtodisplay,init) = cclienthrdpara.update.'regclient.'initmodel.
        'cachtodisplay.'getcacheddata.'init.CLIENTTHRD(regclient,initmodel,
        getcacheddata,cclienthrdpara,update,cachtodisplay,init)
agent CACHEMODEL(initmodel,getcacheddata) =
        initmodel.getcacheddata.CACHEMODEL(initmodel,getcacheddata)
agent TOOL =  USERTABLE | DATASTORE | SERVER | USERMAN | CLIENTMAN |
        UI | CACHEMODEL | UPDATETHRD | CLIENTTHRD
```

The Mobility Workbench analysis result:

```
MWB>deadlocks TOOL
No deadlocks found.
```

6.5 Conclusion

In this chapter, we have examined the use of the Dynamic PARSE Design and Analysis methodology. The PARSE-DAT tool has also been evaluated through four case studies. These case studies are of different architecture types, and vary in complexity. We have shown the expressiveness of the Dynamic PARSE process graph design notation, demonstrated the typical Dynamic PARSE design and analysis processes, as well as the value of the PARSE-DAT design analysis tool.

7. Conclusions

This thesis has presented a novel software architecture design and verification methodology for dynamic distributed systems. Architects employ a pragmatic, graphical design method called Dynamic PARSE (PARSE-D) to design the software architecture, which provides an explicit representation of the parallelism and distribution in a system via a well defined set of model elements. The architects thereby capture the concurrent and dynamic features of the system. Such dynamic features include the creation and deletion of processes and re-configurable communication links. Lastly, the correctness of the design can be verified, and possible design faults may be detected by using an automatic design analysis and verification tool called PARSE-DAT. This integrated rapid prototyping environment with an automated analysis facility enables the designer to verify their design at an early stage of the engineering process, hence reducing resource overheads incurred in discovering deadlock at the later testing stage.

Specifically, the following has been presented in this thesis:

- Dynamic PARSE Process Graph Design Notation: which is an extension to the architecture description language PARSE. This resultant set of notations can describe dynamic features such as the creation and deletion of process components, and dynamic communication reconfiguration.
- Translations of Dynamic PARSE Process Graph Design Notation to two formalisms: π-Calculus and Self-Modifying Nets.
- A distributed software engineering methodology (PARSE-D) supporting design using Dynamic PARSE Process Graph Notation and design analysis/verification utilising π-Calculus.
- An integrated tool-set environment called PARSE-DAT for supporting the PARSE-D software engineering methodology. The two major components are the CASE tool

PARSE-DT that enables the construction of Dynamic PARSE designs; and an automated analysis/verification tool called PARSE-AT.

An empirical study of the 'influence of formal methods' [Pfleeger97] has shown that formal design, combined with other techniques, yields highly reliable code. Further, in a formal methods roundtable presented in the April 1996 IEEE Computer, it was pointed out that: "...the purpose of formalisation is to reduce the risk of serious errors in specification and design. Analysis can expose such errors while they are still cheap to fix".

PARSE-D is precisely a software engineering methodology in this direction.

Future Directions

There are several areas of possible future work to extend the capability of the Dynamic PARSE design and analysis methodology as well as the PARSE-DAT environment.

Machine aided design refinement: regarding the formal design analysis method, there is the possibility of providing support for machine aided design refinement, in order to bridge the gap between the design stage and the implementation stage. There is much sound theoretical π-Calculus work in the areas of equivalence. The work in exploring the possibilities of refining Dynamic PARSE designs through π-Calculus would be interesting and valuable.

Incorporating a Performance Prediction Tool: on the issue of CASE tool support, there is currently independent work done in the area of performance evaluation and prediction [Hu97]. Work in incorporating the performance prediction tool into PARSE-DAT should be fruitful.

Integration with object-oriented design methods: In this thesis, we have described Dynamic PARSE as an architecture design methodology. Once software architecture has been devised and validated, detailed design is the next development stage. Object-oriented methodologies provide a seamless transition from the analysis stage to the design stage. It is anticipated that integrating the Dynamic PARSE methodology with an object-oriented approach at the detailed design stage would be an effective way of developing software. This integration work could take place on two levels. Firstly on the methodology level, mechanisms need to be worked out to support the transition from architecture design in Dynamic PARSE onto detailed object-oriented designs. Secondly, tool support can be carried out by integrating an existing object-oriented CASE tool (such as Rational Rose) into PARSE-DAT.

Detailed design analysis: PARSE-DAT enables automated design analysis and verification on an architectural level. Automated analysis on an implementation/ algorithmic/ code level is worth exploring in the future.

Appendix

A. Sample Mobility Workbench Output

The following is the π-Calculus model of an intermediate HTPNet design that exhibits presence of structural deadlock. The Mobility workbench analysis output is also included.

```
HEADER(data,status,store,st) = data(d).'store<d>.HEADER(data,status,store,st)
                             + status(sr).'sr<st>.HEADER(data,status,store,st)
TEMPFIFO(store,retrieve)                                                      =
store(d).retrieve(req).'req<d>.TEMPFIFO(store,retrieve)
OUTPUT(retrieve,req,recdata) = 'retrieve<req>.req(d).'recdata<d>
                             .OUTPUT(retrieve,req,recdata)
DATAPROC(data,status,recdata) = (^store,retrieve,st,req)
                             (HEADER(data,status,store,st)
                             | TEMPFIFO(store,retrieve)
                             | OUTPUT(retrieve,req,recdata))
SYNKPROC(synk,status,ack,sr) = synk(s).'status<sr>.sr(st).'ack<st>
                                     .SYNKPROC(synk,status,ack,sr)
RECHOST(recdata) = recdata(rd).RECHOST(recdata)
RECEIVE(data,synk,ack) = (^status,recdata,sr)
                             (DATAPROC(data,status,recdata)
                             | SYNKPROC(synk,status,ack,sr)
                             | RECHOST(recdata))
TRANSMIT(data,synk,ack,d,s) = 'data<d>.TRANSMIT(data,synk,ack,d,s)
                             + 'synk<s>.ack(ac).TRANSMIT(data,synk,ack,d,s)
NETWORK(data_t,ack_t,synk_t,data_r,ack_r,synk_r,sr) =
        data_t(d).'data_r<d>.NETWORK(data_t,ack_t,synk_t,data_r,ack_r,synk_r,sr)
        + synk_t(s).ack_t(ac).NETWORK(data_t,ack_t,synk_t,data_r,ack_r,synk_r,sr)
        +'synk_r<sr>.ack_r(ac).NETWORK(data_t,ack_t,synk_t,data_r,ack_r,synk_r,sr
)
SYSTEM = (^ack_t,data_t,synk_t,ack_r,data_r,synk_r,sr_t,sr_n,d)
              (TRANSMIT(data_t,synk_t,ack_t,d,sr_t)
              | NETWORK(data_t,ack_t,synk_t,data_r,ack_r,synk_r,sr_n)
              | RECEIVE(data_r,synk_r,ack_r))
```

```
MWB>deadlocks RECEIVE
No deadlocks found.

MWB>deadlocks TRANSMIT
No deadlocks found.
MWB>deadlocks NETWORK
No deadlocks found.
MWB>time deadlocks SYSTEM
Deadlock found in
(^~v,~v2,~v3,~v4,~v5,~v6,~v7,~v8,~v9)(~v.(\ac)TRANSMIT<~v2,~v3,~v,~v9,~v7> |
(^~v10)(~v.(\ac)N<~v2,~v,~v3,~v5,~v4,~v6,~v8> |
(^~v11,~v12,~v13)((^~v14,~v15,~v16)((~v5.(\d)'~v14.[d]HEADER<~v5,~v11,~v14,~v10>
+ ~v11.(\sr)'sr.[~v10]HEADER<~v5,~v11,~v14,~v10>) |
~v14.(\d)~v15.(\req)'req.[d]TEMPFIFO<~v14,~v15> |
'~v15.[~v16]~v16.(\d)'~v12.[d]OUTPUT<~v15,~v16,~v12>) |
~v6.(\s)'~v11.[~v13]~v13.(\st)'~v4.[st]SYNKPROC<~v6,~v11,~v4,~v13> |
~v12.(\rd)RECHOST<~v12>)))
 reachable by 5 commitments
Deadlock found in
(^~v,~v2,~v3,~v4,~v5,~v6,~v7,~v8,~v9)(~v.(\ac)TRANSMIT<~v2,~v3,~v,~v9,~v7> |
(^~v10)(~v.(\ac)N<~v2,~v,~v3,~v5,~v4,~v6,~v8> |
(^~v11,~v12,~v13)((^~v14,~v15,~v16)((~v5.(\d)'~v14.[d]HEADER<~v5,~v11,~v14,~v10>
+ ~v11.(\sr)'sr.[~v10]HEADER<~v5,~v11,~v14,~v10>) |
~v14.(\d)~v15.(\req)'req.[d]TEMPFIFO<~v14,~v15> |
'~v15.[~v16]~v16.(\d)'~v12.[d]OUTPUT<~v15,~v16,~v12>) |
~v6.(\s)'~v11.[~v13]~v13.(\st)'~v4.[st]SYNKPROC<~v6,~v11,~v4,~v13> |
~v12.(\rd)RECHOST<~v12>)))
 reachable by 11 commitments
Deadlock found in
(^~v,~v2,~v3,~v4,~v5,~v6,~v7,~v8,~v9)(~v.(\ac)TRANSMIT<~v2,~v3,~v,~v9,~v7> |
~v.(\ac)N<~v2,~v,~v3,~v5,~v4,~v6,~v8> |
(^status,recdata,sr)((^store,retrieve,st,req)((~v5.(\d)'store.[d]HEADER<~v5,stat
us,store,st> + status.(\sr17)'sr17.[st]HEADER<~v5,status,store,st>) |
store.(\d)retrieve.(\req18)'req18.[d]TEMPFIFO<store,retrieve> |
'retrieve.[req]req.(\d)'recdata.[d]OUTPUT<retrieve,req,recdata>) |
~v6.(\s)'status.[sr]sr.(\st)'~v4.[st]SYNKPROC<~v6,status,~v4,sr> |
recdata.(\rd)RECHOST<recdata>))
 reachable by 1 commitments
Deadlock found in
(^~v,~v2,~v3,~v4,~v5,~v6,~v7,~v8,~v9)(~v.(\ac)TRANSMIT<~v2,~v3,~v,~v9,~v7> |
~v.(\ac)N<~v2,~v,~v3,~v5,~v4,~v6,~v8> |
(^~v10,~v11,~v12)((^~v13,~v14,~v15,~v16)((~v5.(\d)'~v13.[d]HEADER<~v5,~v10,~v13,
~v15> + ~v10.(\sr)'sr.[~v15]HEADER<~v5,~v10,~v13,~v15>) |
~v13.(\d)~v14.(\req)'req.[d]TEMPFIFO<~v13,~v14> |
'~v14.[~v16]~v16.(\d)'~v11.[d]OUTPUT<~v14,~v16,~v11>) |
~v6.(\s)'~v10.[~v12]~v12.(\st)'~v4.[st]SYNKPROC<~v6,~v10,~v4,~v12> |
~v11.(\rd)RECHOST<~v11>))
 reachable by 7 commitments
```

```
User CPU time elapsed: 18.008499
System CPU time: 2.503421
GC time: 7.274832
Real time elapsed: 71.444265
Overhead: 43.657513
```

Bibliography

[Andersen94] H.R.Andersen (1994) Model Checking and Boolean Graphs, TCS, vol.126, no.1.

[Ansart89] J.P.Ansart, V.Chari (1989) *General Survey on the Estelle Results*, in the book: The Formal Description Technique Estelle, North-Holland, pp.17-34.

[Aonix98] Aonix (1998) *Software Modeling Solutions - Software Through Pictures*, http://www.methods-tools.com/tools/frames_analdes.html

[AST98] Advanced Software Technologies Inc. (1998) *GDPro: Product Overview*, http://www.advancedsw.com/overview.html

[BBN98] BBN (1998) *Distributed Planning & Operations Management - Corbus*, http://www.bbn.com/products/dpom/corbus.htm.

[B-Core96] B-Core UK Limited (1996) *The B Toolkit*, user manual, March 6, 1996.

[Belina91] F.Belina, D.Hogrefe, A.Sarma (1991*) SDL With Applications From Protocol Specification*, Prentice Hall.

[Bhat95] G.Bhat, R.Cleaveland, O.Grumberg (1995) Efficient On-The-Fly Model Checking for CTL*, Proceedings 10th Annual Symposium on Logic in Computer Science (LICS'95), San Diego, July, Computer Science Press, pp.388-97.

[Birkinshaw95] C.I.Birkinshaw, P.R.Croll (1995) *Modelling the Client-Server Behaviour of Parallel Real-Time Systems Using Petri Nets*, Proc. 28th Ann. Hawaii Int'l Conf. System Sciences, Parallel Software Engineering Minitrack, Vol.2: Software Technology, IEEE Computer Society Press, Calif., 339-48.

[Booch94] G. Booch (1994) *Object-Oriented Analysis and Design with Applications,* Redwood City CA: Benjamin Cummings.

[Bowen95] J.P.Bowen and M.G.Hinchey (1995*) Seven More Myths of Formal Methods*, IEEE Software July 1995 vol.12 no.3, pp.34-41.

[Butler96] M.Butler, M.Walden (1996) *Distributed System Development in B*, Proceedings 1st Conference on the B Method, November 24-26, Nantes, France, pp.155-90.

[Cardelli98] L.Cardelli, A.D.Gordon (1998) *Mobile Ambients*, ETAPS'98, also available at http://www.luca.demon.co.uk/

[Carrington94] D.Carrington, I.Hayes, R.Nickson, G,Watson, J.Welsh, *A Review of Existing Refinement Tools*, Technical Report UQ-SVRC-94-8, Software Verification Research Centre, University of Queensland.

[Carter96] F.Carter, A.Fekete (1996) *Cerberus - A Tool For Debugging Distributed Algorithms*, Procedings 1st IFIP TC10 International Workshop on Parallel and Distributed Software Engineering, Chapman and Hall, pp.110-21.

[Cayenne98] Cayenne Software (1998) *Class Designer: New, User-Friendly Tool for Designing Java and C++ Objects*, http:// www.cayennesoft.com/classdesigner/

[Chan93] T.S.Chan, I.Gorton (1993) *A Transputer-based Implementation of HTPNET: a Transport Protocol for Broadband Networks*, in

Transputer Applications and Systems Vol. 2, Proceedings of the 1993 World Transputer Congress, Aachen, Germany, pp 899-910, IOS Press, September.

[Chan94] T.S.Chan and I.Gorton (1994) *A parallel approach to high-speed protocol processing*, in Transputer Applications and Systems 94, Proceedings of the 2nd World Transputer Congress, Como, Italy, September, pp 209-22, IOS Press

[Chappell96] D.Chappell (1996) *Understanding ActiveX and OLE*, Redmond, WA: Microsoft Press, 1996.

[Cheung94] S.C.Cheung, J.Kramer (1994) *Tractable Dataflow Analysis for Distributed Systems*, IEEE Transactions on Software Engineering, vol.20, no.8, August, pp.579-93.

[Cheung94a] S.C.Cheung, J.Kramer (1994) *An Integrated Method for Effective Behaviour Analysis of Distributed Systems*, Proceedings of 16th IEEE International Conference on Software Engineering, Sorrento, Italy, May.

[Ciardo94] G.Ciardo, R.German, C.Lindemann (1994) A Characterization of the Stochastic Process Underlying a Stochastic Petri Net, *Transactions on Software Engineering*, vol.20, pp.506-15.

[Coad91] P.Coad, E.Yourdon (1991) *Object-Oriented Analysis.* Prentice Hall.

[Coad91a] P.Coad, E.Yourdon (1991) *Object-Oriented Design.* Prentice Hall.

[Coleman94] D.Coleman, P.Arnold, S.Bodoff, C.Dollin, H.Gilchrist, F.Hayes, P.Jeremes (1994) *Object-Oriented Development: The Fusion Method,* Upper Saddle River, NJ: Prentice-Hall.

[Comerford94] R.Comerford (1994) *Engineering Workstations and PCs*, IEEE Spectrum, April, pp 45-46.

[Custer93]	H.Custer (1993) *Inside Windows NT*, Microsoft Press.
[Dehbonei95]	B.Dehbonei, F.Mejia (1995) *Formal Development of Safety-Critical Software Systems in Railway Signalling*, in Applications of Formal Methods, edited by Michael G. Hinchey and Jonathan P.Bowen, Prentice-Hall, pp.227-52.
[Diaz89]	M.Diaz, J-P.Ansart, P.Azema, V.Chari (1989) *The Formal Description Technique Estelle*, North-Holland.
[Durr95]	E.H.Durr, N.Plat and M.de Boer (1995*) CombiCom: Tracking and Tracing Rail Traffic using VDM++,* in Applications of Formal Methods, edited by Michael G. Hinchey and Jonathan P.Bowen, Prentice-Hall, pp.203-26.
[Elseaidy96]	W.Elseaidy, R.Cleaveland, J.Baugh (1996) *Modeling and Verifying Active Structural Control Systems*, Science of Computer Programming. To appear. A preliminary version of this paper appears in the Proceedings of the 1994 Real-Time Systems Symposium.
[Eriksson98]	H-E.Eriksson, M.Penker (1998) *UML Toolkit*, Wiley.
[Eucalyptus97]	http://www-run.montefiore.ulg.ac.be/projects/ EUCALYPTUS.html (Jan 1997).
[Faergemand91]	O.Faergemand, R.Reed (1991) *SDL'91 Evolving Methods*, Proc. SDL Forum '91, North-Holland, Amsterdam.
[Faergemand93]	O.Faergemand, A.Olsen (1993*) Tutorial on New Features in SDL-92*, TFL, Telelcommunications Research Laboratory, Lyngso Alle 2, DK-2970 Horsholm Denmark.
[Faergemand94]	O.Faergemand, A.Olsen (1994) *Introduction to SDL-92*, Computer Networks and ISDN Systems vol.26, pp.1129-42.

[Fields92] R.Fields, M.Elvang-Goransson (1992) *A VDM Case Study in Mural*, IEEE Transactions on Software Engineering, vol.18, no.4, April, pp.279-95.

[FME98] Formal Methods Europe (1998) *FME Tools Database - Name of Tool: ProofPower*, http://www.csr.ncl.ac.uk/projects/FME/InfRes/tools/fmtdb027.html

[Gamma95] E.Gamma, R.Helm, R.Johnson, J.Vlissides (1995) *Design Patterns: Elements of Reusable Object-Oriented Design*, Addison-Wesley.

[Garland94] D.Garlan, R.Allen, J.Ockerbloom (1994) *Exploiting Style in Architectural Design Environments*, Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineerng, December.

[Garland95] D.Garlan, D.Perry (1995) *Introduction to the Special Issue on Software Architecture*, IEEE Transactions on Software Engineering, April.

[Gorton93] I.Gorton, I.E.Jelly, J.Gray (1993) *Parallel Software Engineering with PARSE*, in Proceedings of COMPSAC-17, IEEE Int. Computer Software and Applications Conference, November, Phoenix, Arizona, USA, IEEE.

[Gorton94] I.Gorton, T.S.Chan, I.E.Jelly (1994) *Engineering High Quality Parallel Software Using PARSE*, Parallel Processing: CONPAR 94 - VAPP VI, pp.381-92.

[Gorton95] I.Gorton, J.Gray, I.E.Jelly (1995) *Object-based Modelling Of Parallel Programs,* IEEE Parallel and Distributed Technology, Summer edition, pp.52-63.

[Gorton95a] I.Gorton, I.E.Jelly, P.Croll, P.Nixon (1995) *Directions in Software Engineering for Parallel Systems*, Proceedings of 28th Hawaii

	International Conference on System Sciences, Software Technology Track, Jan 3-6, IEEE.
[Gorton96]	I.Gorton, I.E.Jelly, J.Gray, T.S.Chan (1996) *Reliable Parallel Software Construction Using PARSE*, Concurrency: Practice and Experience, vol.8(2), pp125-46, March.
[Gorton96a]	I.Gorton, J.Zic, I.E.Jelly (1996) *Supporting Multiple Formal Methods in PARSE*, University of New South Wales, internal report.
[Gorton96b]	I.Gorton, Y.H.Ng, A.Liu (1996*) Generating Occam From PARSE Process Graphs*, Transputer Communications, vol.3, no.1, pp.51-9
[Gorton97]	I.Gorton, I.T.Hawryszkiewycz, C.Chung, S.Lu, K.Ragoonaden (1997) *Groupware Support Tools for Collaborative Software Engineering*, 30th Hawaii International Conference on System Sciences, Informations Systems Track, IEEE, Hawaii January.
[Gorton97a]	I.Gorton, I.Jelly (1997) Software Engineering for Parallel and Distributed Systems: Challenges and Opportunities, IEEE Concurrency, vol.5, no.3, pp.12-15.
[Gray94]	J.Gray (1994) *Definition of the PARSE Process Graph Notation, version 2 (PGN/2),* Technical Report PARSE-TR-2b, March 1994.
[Gray97]	J.P.Gray, B.Ryan (1997) *Applying the CDIF standard in the Constructing of CASE Design Tools,* In Proceedings of Australian Software Engineering Conference (ASWEC97), Sydney Australia, 28 Sept - 3rd Oct, IEEE Computer Society Press.
[Harel88]	D.A.Harel (1988) *On visual formalisms*, Communications of ACM, vol.31, no.5.
[Harel90]	D.Harel, H.Lachover, A.Naamad, A.Pnueli, M.Politi, R.Sherman,

	A.Shtull-Trauring, M.Trakhtenbrot (1990) *Statemate: A Working Environment for the Development of Complex Reactive Systems*, Transactions on Software Engineering, vol.16, no.4, pp.403-414.
[Harel97]	D.Harel, E.Gery (1997) *Executable Object Modeling with Statecharts*, IEEE Computer, July, pp.31-42.
[Harel-E90]	E.Harel, O.Lichtenstein, A.Pnueli (1990) *Explicit Clock Temporal Logic*, Proceedings of 5th Annual Symposium on Logic in Computer Science, June, pp.402-13.
[Hawryszkiewycz96]	I.T.Hawryszkiewycz, I.Gorton (1996) *Distributing the Software Process*, Australian Software Engineering Conference, Melbourne, Australia, pp.176-82.
[Hayes91]	I.Hayes (1991) *Applying Formal Specification to Software Development in Industry*, in Specification Case Studies, Prentice-Hall, pp.285-310.
[Hayes91a]	I.Hayes (1991) *CICS Temporary Storage*, in Specification Case Studies, Prentice-Hall, pp.311-24.
[Hayes91b]	I.Hayes (1991) *CICS Message System*, in Specification Case Studies, Prentice-Hall, pp.325-32.
[Henderson92]	B.Henderson-Sellers (1992) *A Book of Object-Oriented Knowledge*, Prentice Hall.
[Hoare69]	C.A.R.Hoare (1969) *An Axiomatic Basis for Computer Programming*, Communications of the ACM, vol.12, no.10, October.
[Hoare85]	C.A.R.Hoare (1985) *Communicating Sequential Processes*, Prentice Hall.

[Hoare-JP95] J.P.Hoare (1995) *Application of the B-Method to CICS*, in Applications of Formal Methods, edited by M.G.Hinchey and J.P.Bowen, Prentice-Hall, pp.97-124.

[Holloway96] C.M.Holloway, R.W.Butler (1996) *Impediments to Industrial Use of Formal Methods, Formal Methods Roundtable*, IEEE Computer, April, pp.25-6.

[Hooman89] J.Hooman, W-P.deRoever (1989) *Design and Verification in Real-Time Distributed Computing: An Introduction to Compositional Methods*, Proceedings of 9th International Symposium on Protocol Specification, Testing and Verification, North-Holland.

[Horstmann98] C.S.Horstmann, G.Cornell (1998) *Core Java 1.1 Volume 1 - Fundamentals*, Sun Microsystems Press.

[Howland-Rose94] K.Howland-Rose, U.Szewcow (1994) *Exposing Concurrency, an Object Oriented Approach - Objects to Occam*, IOS TCAS Conference.

[HP96] Hewlett-Packard (1996) *HP ORB Plus 2.0 For HP-UX, SunSoft Solaris, and Microsoft Windows NT - Product Brief*, http://www.hp.com/gsy/orbplus.html, updated May 22.

[Hu97] L.Hu, I.Gorton (1997) *A Performance prototyping Approach to Designing Concurrent Systems*, Proceedings of 2nd International Workshop on Software Engineering for Parallel and Distributed Systems, IEEE Computer Society Press, California, pp.270-6.

[Iona95] Iona Technologies Ltd. (1995) *Orbix 2 - Distributed Object technology, Programming and Reference Guide*, Release 2.0, Iona Technologies Ltd.

[Jacobson92] I.Jacobson, M.Chriterson, P.Jonsson, G.Overgaard (1992*) Object-Oriented Software Engineering*, Reading NY: Addison-Wesley, 1992.

[Jacobson92a] I.Jacobson (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

[Jahanian88] F.Jahanian, D.Stuart (1988) *A Method for Verifying Properties of Modechart Specifications*, Proceedings 9th Real-Time Systems Symposium, IEEE Computer Society, December, pp.12-21.

[Jelly95] I.Jelly, S.Russo, C.Savy (1995) From Textual Representation of PARSE Designs to Petri Nets, Research Note - Draft version, April 1995, through personal communication.

[Jelly96] I.Jelly, I.Gorton (1996) *Current Research Directions in Software Engineering for Parallel and Distributed Systems*, Software Engineering Notes, through personal communication.

[Jelly96a] I.E.Jelly, I.Gorton (1996) *Case Tools For Parallel Systems*, Transputer Communicatioins, 1996, vol.3, no.1, pp.3-6.

[Jensen92] K.Jensen (1992) *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Volume 1, Basic Concepts, Monographs in Theoretical Computer Science, Springer-Verlag.

[Jones89] C.B.Jones (1989) *Systematic Software Development Using VDM*, 2nd Edition, Englewood Cliffs, Prentice Hall.

[Jones91] C.B.Jones, K.D.Jones, P.A.Lindsay, R.Moore (1991) *MuRAL: A Formal Development Support System*, Springer-Verlag.

[Klein93] M.H.Klein, T.Ralya, B.Pollak, R.Obenza, M.G.Harobur (1993) *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*, Kluwer Academic.

[Koymans90] R.Koymans (1990) *Specifying Real-Time Properties with Metric*

	Temporal Logic, Real-Time Systems, vol.2, no.4, November, pp.255-99.
[Kramer85]	J.Kramer, J.Magee (1985) *Dynamic Configuration for Distributed Systems*, IEEE Transactions on Software Engineering, vol.SE-11, no.4, April 1985, pp.424-35.
[Kramer94]	J.Kramer (1994) *Distributed Software Engineering*, Proceedings 16th International Conference on Software Engineering, IEEE Computer Society Press, California, pp.253-63.
[Kurshan94]	R.P.Kurshan (1994) *Computer-Aided Verification of Coordinating Processes*, Princeton University Press.
[Lakos91]	C.Lakos (1991) *Simulation with Object-Oriented Petri Nets*, Proceedings Australian Software Engineering Conference, Sydney, Australia, July.
[Lano96]	K.Lano (1996) *The B Language and Method: A Guide To Practical Formal Development*, Springer-Verlag London.
[Larocque94]	J.Larocque (1994) *Client-Server Trend,* IEEE Spectrum, April, pp.48-50.
[Lea93]	R.Lea, C.Jacquemot, E.Pillevesse (1993) *COOL: System Support for Distributed Object-Oriented Programming*, Chorus Systems, September.
[Leduc94]	G.Leduc (1994) *A Method for Applying LOTOS at an Early Design Stage and its Application to the ISO Transport Protocol*, The OSI95 Transport Service with Multimedia Support, Springer-Verlag, Berlin, 151-80.
[Liu73]	C.L.Liu, J.W.Layland (1973) *Scheduling Algorithm for Multiprogramming in a Hard Real-Time Environment*, Journal of

the ACM, vol.20, no.1, January, pp.46-61.

[Liu96] A.Liu, I.Gorton (1996) *Modelling Dynamic Distributed System Structures in PARSE*, Proceedings of 4th Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, California, 352-9.

[Liu96a] A.Liu, T.S.Chan, I.Gorton (1996*) Designing Distributed Multimedia Systems Using PARSE*, First IFIP Workshop on Software Engineering for Parallel and Distributed Systems, Chapman and Hall, Berlin, Germany, 25-26 March, pp.50-61.

[Liu97] A.Liu, I.Gorton, (1997) *Designing Distributed Object Systems with PARSE*, Proceedings of 5th Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, California, pp.335-42.

[Liu97a] A.Liu, I.Gorton, J.Zic (1997) *Formalising PARSE Design Notations with π-Calculus*, internal report, Dept. of Computer Systems, School of Computer Science and Engineering, University of New South Wales.

[Liu98] A.Liu, I.Gorton (1998) PARSE-DAT: An Integrated Environment for the Design and Analysis of Dynamic Software Architectures, Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems, IEEE Computer Society Press, California, pp.146-55.

[Low96] G.C.Low, G.Rasmussen, B.Henderson-Sellers (1996) *Incorporation of Distributed Computing Concern into Object-Oriented Methodologies*, Journal of Object-Oriented Programming, June, pp.12-20.

[Luckham95] D.C.Luckham, J.Vera (1995*) An Event-Based Architecture Definition Language*, IEEE Transactions on Software Engineering, vol.21, no.9, pp.717-34, September.

[Luckham95a] D.Luckham, J.J.Kennedy, L.M.Augustin, J.Vera, D.Bryan, W.Mann (1995) *Specification and Analysis of System Architecture Using Rapide*, IEEE Transactions on Software Engineering, special issue on Software Architecture, vol.24, no.4, pp.336-55, April.

[Magee89] J.Magee, J.Kramer, and S.Sloman, (1989) *Constructing Distributed Systems in Conic*, IEEE Transaction on Software Engineering, vol.15, pp.663-75.

[Magee94] J.Magee, N.Dulay, J.Kramer (1994) Regis: A Constructive Development Environment for Distributed Programs. In IEE/IOP/BCS Distributed Systems Engineering, vol.1, no.5, pp.304-12, September.

[Magee95] J.Magee, N.Dulay, S.Eisenbach, J.Kramer (1995) *Specifying Distributed Software Architectures*, Proceedings of 5th European Software Engineering Conference (ESEC 95), Sitges, Spain, September.

[MetaCase96] MetaCase Consulting (1996) *MetaEdit+ version 2.5, User's Guide*, MicroWorks, Finland.

[MetaCase96a] MetaCase Consulting (1996) *MetaEdit+ version 2.5, Method Workbench User's Guide*, MicroWorks, Finland.

[MetaCase96b] MetaCase Consulting (1996) *MetaEdit+ version 2.5, System Administrator's Guide*, MicroWorks, Finland.

[MetaCase96c] MetaCase Consulting (1996) *MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment*, White paper, February, Finland.

[MetaCase96d] MetaCase Consulting (1996) *Developing New Methods with the*

MetaEdit Personal Environment, White paper, February, Finland.

[Microsoft98]	Microsoft (1998) Microsoft COM Home - Component Object Model, last updated July 2, http://www.microsoft.com/com/
[Milner89]	R.Milner (1989) *Communication and Concurrency*, Prentice Hall.
[Milner91]	R.Milner (1991) *The Polyadic π-Calculus: a Tutorial*, Laboratory for foundations of Computer Science, Computer Science Department, University of Edinburgh.
[Milner92]	R.Milner, J.Parrow, D.Walker (1992) *A Calculus of Mobile Processes - Part I,* Information and Computation, Vol.100, pp.1-40.
[Milner92a]	R.Milner, J.Parrow, D.Walker (1992) *A Calculus of Mobile Processes - Part II,* Information and Computation, Vol.100, pp.41-77.
[Mok91]	A.K.Mok (1991) *Towards Mechanisation of Real-Time System Design*, In Foundations of Real-Time Computing: Formal Specifications and Methods, Kluwer Press.
[Monroe96]	R.T.Monroe (1996) *Capturing Design Expertise in Customized Software Architecture Design Environments*, Proceedings of the Second International Software Architecture Workshop, October.
[Monroe97]	R.T.Monroe, A.Kompanek, R.Melton, D.Garlan (1997) *Architectural Styles, Design Patterns, and Objects*, IEEE Software January, pp.43-52.
[Morgan90]	C.C.Morgan (1990) *Programming from Specifications*, Prentice-Hall.
[Nairn96]	G.Nairn (1996) *Complexity Drives Search for Simple, Fail-safe Software*, in The Australian Newspaper, Tuesday January 16.

[Ng95] K.Ng, J.Kramer (1995) *Automated Support for Distributed Software Design*, Proceedings of 7th International Workshop on Computer-aided Software Engineering (CASE 95), Toronto, Canada, July.

[Nierstrasz92] O.Nierstrasz, S.Gibbs, D.Tsichritzis (1992) *Component-Oriented Software Development*, Communication of ACM, vol.35, no.9.

[OMG95] Object Management Group (1995) *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, OMG.

[Oreizy98] P.Oreizy, N.Medvidovic, R.N.Taylor (1998) *Architecture-Based Runtime Software Evolution*, Proceedings of the International Conference on Software Engineering (ICSE'98). Kyoto, Japan, April 19-25.

[Ostroff85] J.S.Ostroff, W.M.Wonham (1985) *A Temporal Logic Approach to Real Time Control*, Proceedings of 24th IEEE Conference on Decision and Control, Florida, December, pp.656-7.

[Pateman95] S.Pateman (1995) *An Investigation of PARSE and DISC Integration*, Computing Research Centre, Sheffield Hallam University, Technical Report Series CRC-95-6.

[Pecheur92] C.Pecheur (1992) *Using LOTOS for specifying the CHORUS distributed operating system kernel*, Computer Communications, vol.15, no.2, March, pp.93-102.

[Peled96] D.Peled (1996) *Combining Partial Order Reductions With On-The-Fly Model-Checking*, Journal of Formal Methods in Systems Design, vol.8, no.1, pp.39-64.

[Peterson81] J.L.Peterson (1981) *Petri net Theory and the Modeling of Systems*, Englewood Cliffs, N.J., Prentice-Hall.

[Petri62] C.A.Petri (1962) *Kommunikation mit Automaten*, Schriften des Iim 2, Institut fur Instrumentelle Mathematik, Bonn. English translation available as *Communication with Automata*, Technical Report RADC-TR-65-377, vol.1 supplement.1, Applied Data Research, Princeton, NJ, 1966.

[Pfleeger97] S.L.Pfleeger, L.Hatton (1997) *Investigating the Influence of Formal Methods*, IEEE Computer, February, pp.33-43.

[Pnueli77] A.Pnueli (1977) *The Temporal Logic of Programs*, Proceedings 18th IEEE Symposium on Foundations of Computer Science, Computer Society Press, pp.46-57.

[Potter91] B.Potter, J.Sinclair, D.Till (1991) *An Introduction to Formal Specification and Z*, Prentice-Hall, C.A.R. Hoare Series Editor.

[Quatrani98] T.Quatrani (1998) *Visual Modeling with Rational Rose and UML*, Addison-Wesley.

[Rashid89] R.Rashid, R.Baron, A.Forin, D.Golub, M.Jones, D.Julin, D.Orr, R.Sanzi (1989) *Mach - A Foundation to Open Systems*, Proceedings of the 2nd Workshop on Workstation Operating Systems, September.

[Rasmussen96] G.Rasmussen, B.Henderson-Sellers, G.C.Low (1996) *Extending the MOSES Methodology to Distributed Systems*, Journal of Object-Oriented Programming, July-August, pp.39-46.

[Renesse89] R.vanRenesse, H.vanStaveren, A.S.Tanenbaum (1989) *The Performance of the Amoeba Distributed Operating System*, Software - Practice and Experience, vol.19, March, pp.223-34.

[Rice94] M.D.Rice, S.B.Seidman (1994) *A Formal Model for Module Interconnection Languages*, IEEE Transactions on Software

Engineering, vol.20, no.1, January, pp.88-101.

[Richter94] J.Richter (1994) *Advanced Windows NT, The Developer's Guide to the Win32 Application Programming Interface*, Microsoft Press.

[Roscoe97] A.W.Roscoe (1997) *The Theory and Practice of Concurrency*, Prentice-Hall International Series in Computer Science.

[Rozier90] M.Rozier, V.Abrossimov, F.Armand, J.Boule, M.Gien, M.Guillemont, F.Herrmann, C.Kaiser, S.Langlois, P.Leonard, W.Neuhauser (1990) *Overview of the CHORUS Distributed Operating System*, Chorus Systems.

[Rumbaugh91] J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, F.Lorensen (1991) *Object-Oriented Modeling and Design*, Englewood Cliffs NJ: Prentice-Hall.

[Saiedian96] H.Saiedian (1996) *An Invitation to Formal Methods*, IEEE Computer, April, pp.16-30

[Sangiorgi96] D.Sangiorgi (1996) Bisimulation in Higher-Order Process Calculi, Journal of Information and Computation, vo.131, pp.141-78.

[Saracco89] R.Saracco, R.Reed, J,Smith (1989) *Telecommunications Systems Engineering*, North-Holland Elsevier.

[Shan95] Y-P.Shan, R.Earle, S.McGaughey (1995) *Distributed Objects - Rounding Out the Picture: Objects Across the Client-Server Spectrum, Object Technology - A Virtual Roundtable*, IEEE Computer, October, pp.60.

[Shaw96] M.Shaw, D.Garland (1996) *Software Architecture: Perspective on an Emerging Discipline*, Prentice-Hall, New Jersey.

[Shlaer93] S.Shlaer, S.J.Mellor (1993) A deeper look at the transition from analysis to design, Journal of Object-Oriented Programming, February.

[Siemen97] N.Richter (1997) *DCE - Product Families*, Siemens Nixdorf Informations systeme AG, Last Update: August 12, 1997, http://www.sni.de/servers/dce/dce_us/dceprod.htm

[Silberchatz91] Silberchatz, J.Peterson, P.Galvin (1991) *Operating System Concepts,* 3rd ed., Addison-Wesley.

[Spivey88] J.M.Spivey (1988) *Understanding Z, A Specification Language and Its Formal Semantics*, Cambridge UK: Cambridge University Press.

[Spivey92] J.M.Spivey (1992) *The Z Notation: A Reference Manual, 2nd Edition*, Prentice Hall, New York.

[Spragins92] J.D.Spragins, J.L.Hammond, K.Pawlikowski (1992) *Telecommunications - Protocols and Design*, Addison-Wesley.

[Stevens98] P.Stevens (1998) *The Edinburgh Concurrency Workbench,* http://www.dcs.ed.ac.uk/home/cwb/

[Stovsky88] M.P.Stovsky, B.W.Weide (1988) *Building Interprocess Communication Models Using STILE*, Proceedings 21st Annual Hawaii International Conference On Systems Sciences, vol.2, pp.639-47.

[Sun98] Sun Microsystems (1998) *Software & Networking - Solaris Products*, http://www.sun.com/solaris/datasheets/ ds-solaris-os.html

[Taylor96] R.N.Taylor, N.Medvidovic, K.M.Anderson, E.J.Whitehead Jr., J.E.Robbins, K.A.Nies, P.Oreizy, D.L.Dubrow (1996) *A Component- and Message-Based Architectural Style for GUI Software*, IEEE Transactions on Software Engineering, June.

[Transarc98] Transarc (1998) *DCE Product Overview - Building Effective Distributed Computing Solutions*, http://www.transarc.com/dfs/public/www/htdocs/.hosts/external/Product/Txseries/DCE/DCEOverview/dceoverview.html

[Valk77] R.Valk (1977) *Self-modifying Nets*, Technical report no.34, University of Hamburg, Germany, July.

[Valk77a] R.Valk (1997) *Generalizations of Petri Nets.*

[Valk78] R.Valk (1978) *On The Computational Power Of Extended Petri Nets*, Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science, LNCS, vol.64, Berlin: Springer-Verlag, September, pp.526-35.

[Valk93] R.Valk (1993) *Extending S-invariants for Coloured and Selfmodifying Nets*, Technical report no.165, University of Hamburg, Germany, December.

[Valmari96] A.Valmari (1996) *Verification Algorithm Research Group at Tampere University of Technology*, Tietojenkäsittelytiede, no.8, pp. 25-38. Also available at http://www.cs.tut.fi/ohj/VARG/VARG.html

[Victor94] B.Victor (1994) *A Verification Tool for the Polyadic π-Calculus*, Licentiate Thesis, Technical report DoCS 94/50, Dept. of Computer Systems, Uppsala University. May be obtained via ftp://ftp.docs.uu.se/pub/mwb/

[Wirfs90] R.J.Wirfs-Brock, B.Wilkerson, L.Wiener (1990) *Designing Object-Oriented Software,* Prentice Hall.

[Wordsworth96] J.B.Wordsworth (1996) *Software Engineering with B: An Introduction*, Addison-Wesley.

[Yourdon79]	E.Yourdon, L.Constantine (1979) *Structured Design*, Prentice Hall.
[Yourdon89]	E.Yourdon (1989*) Modern Structured Analysis*, Prentice Hall.