MODEL DRIVEN COMMUNICATION PROTOCOL ENGINEERING AND SIMULATION BASED PERFORMANCE ANALYSIS USING UML 2.0

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,

FACULTY OF SCIENCE

AT THE UNIVERSITY OF CAPE TOWN

IN FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Nico de Wet

December 2004

Supervised by

Prof. Pieter S. Kritzinger

Abstract

The automated functional and performance analysis of communication systems specified with some Formal Description Technique has long been the goal of telecommunication engineers. In the past SDL and Petri nets have been the most popular FDTs for the purpose. With the growth in popularity of UML the most obvious question to ask is whether one can translate one or more UML diagrams describing a system to a performance model. Until the advent of UML 2.0, that has been an impossible task since the semantics were not clear. Even though the UML semantics are still not clear for the purpose, with UML 2.0 now released and using ITU recommendation Z.109, we describe in this dissertation a methodology and tool called proSPEX (protocol Software Performance Engineering using XMI), for the design and performance analysis of communication protocols specified with UML.

Our first consideration in the development of our methodology was to identify the roles of UML 2.0 diagrams in the performance modelling process. In addition, questions regarding the specification of non-functional duration contraints, or temporal aspects, were considered. We developed a semantic time model with which a lack of means of specifying communication delay and processing times in the language are addressed. Environmental characteristics such as channel bandwidth and buffer space can be specified and realistic assumptions are made regarding time and signal transfer.

With proSPEX we aimed to integrate a commercial UML 2.0 model editing tool and a discrete-event simulation library. Such an approach has been advocated as being necessary in order to develop a closer integration of performance engineering with formal design and implementation methodologies. In order to realize the integration we firstly identified a suitable simulation library and then extended the library with features required to represent high-level SDL abstractions, such as extended finite state machines (EFSM) and signal addressing. In implementing proSPEX we filtered the XML output of our editor and used

text templates for code generation. The filtering of the XML output and the need to extend our simulation library with EFSM abstractions was found to be significant implementation challenges.

Lastly, in order to to illustrate the utility of proSPEX we conducted a performance analysis case-study in which the efficient short remote operations (ESRO) protocol is used in a wireless e-commerce scenario.

Acknowledgements

The research project resulting in this dissertation has been challenging and rewarding. I learned a great deal about the field of protocol performance engineering and also enhanced my time management skills and learned to focus on the important aspects of a vast body of knowledge. I would like to thank the following people for their support that helped me in completing this work.

- My supervisor, Professor Pieter Kritzinger, for his support and guidance throughout the project.
- Fellow students of the DNA and CVC lab. In particular Ben Tobler, Simon Lukell, Oksana Ryndina, Johannes Appenzeller, Mwelwa Chibesakunda, LourensWalters and Barry Steyn for being great friends and your company. I will miss you!
- My mom and dad, Tibbie and Dirk de Wet, for your unconditional love and support.
- Marelize Retief, for your love, encouragement and always being there for me.
- Fellow M.Sc. students and staff at our department for your advise and assistance. In particular I would like to thank Andrew Hutchinson, Justin Kelleher and Eve Gill.
- The open-source community, in particular to those working on the Apache Jakarta projects, for freely distributing their tools.
- The South African National Research Foundation, for funding this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Definition and Motivation

The problem of designing efficient, correct and unambiguous computer network communication protocols has existed for over five decades. Problems encountered in contemporary protocol design were encountered when designing the first master-slave protocols, used by mainframes in the 1950s, and peer protocols, used in the first large-scale computer networks in the 1960s. One of the fundamental problems protocol engineers have been faced with is how to design and implement large sets of rules for data exchange that are minimal, logically consistent, complete and efficiently implemented[Hol91]. Over the last five decades significant strides have been made towards the resolution of the fundamental problems encountered in protocol design and implementation, leading to the field of *protocol engineering*[Sal96]. The protocol engineering process, with its goal of efficient and reliable software, is an interdisciplinary engineering process which encompasses the application of formal description techniques (FDTs) and sound software engineering methodologies.

The primary steps[MGS+00] in protocol engineering, as shown in Figure 1, are:

- **Formal Specification**. A formal service and protocol description (or validation model) is made from an informal description. The informal description is in English prose in a similar fashion to Request For Comments (RFC) protocol specifications.
- **Validation and verification**. Tools are used for exhaustive reachability analysis.
- **Performance analysis**. Analytical or empirical methods are used to analyse the protocol allowing performance optimization. Non-functional properties regarding time

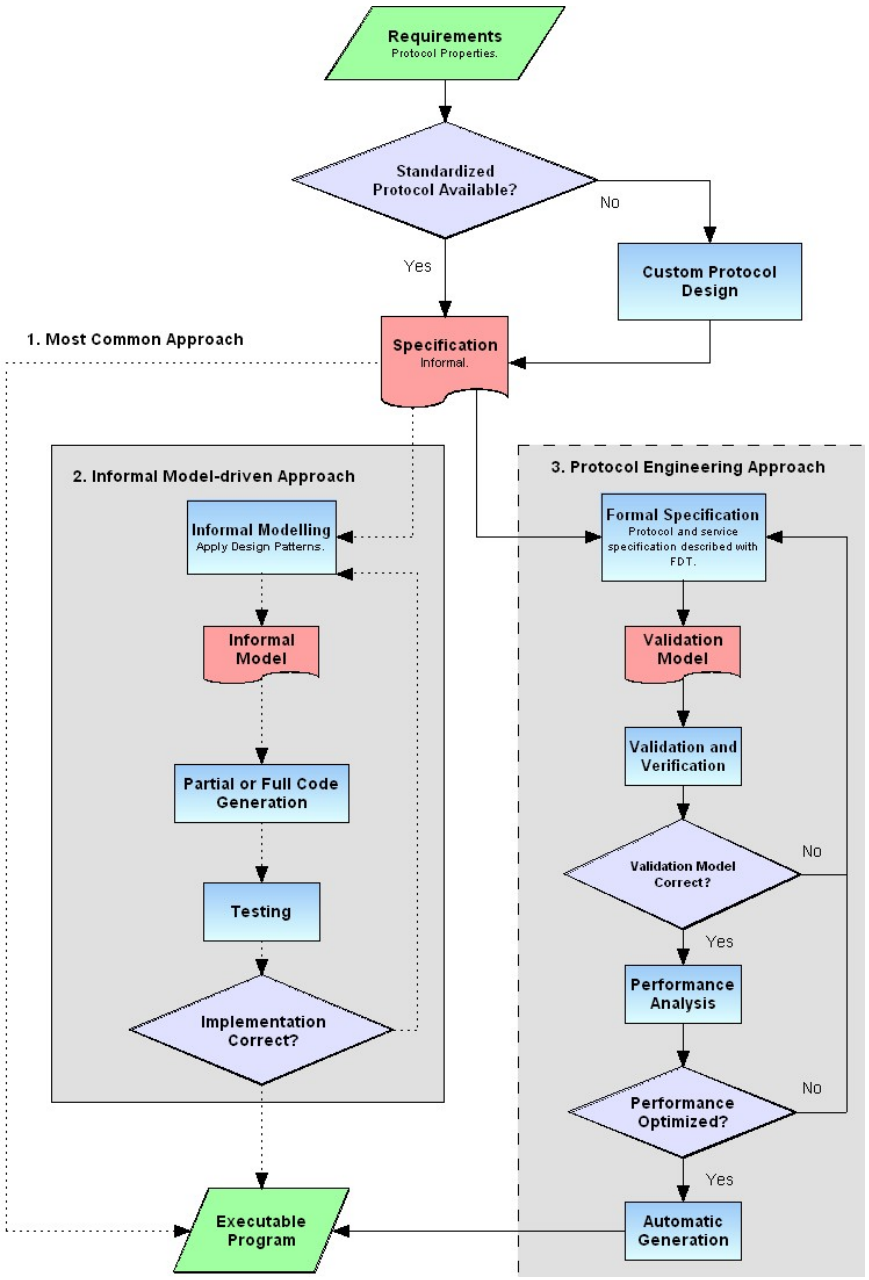

Figure 1: The protocol engineering process.

and resources must be added to support the automatics construction of a performance model[MTMC99].

- **Implementation generation**. Executable code is generated using the validation model.

- **Conformance testing**. Correctness and conformance testing is conducted to determine whether the implementation adheres to the specification.

Several FDTs and associated model checking tools exists to support the first two steps in the protocol engineering process. The most prominent of these FDTs are the Process Meta Language (PROMELA), the Specification and Description Language (SDL) and Estelle. While it is generally accepted that protocols should be specified using such formal languages, thereby allowing validation and verification, less of an emphasis is placed on *design* and *performance analysis*. Protocol design, in particular the architectural design, is know to directly affect a number of software attributes including performance[MTMC99]. Performance is a fundamental attribute of any software which is often neglected in the software engineering life-cycle. In the context of developing performance-critical applications using SDL, Mitschele-Thiel[MTMC99] states:

> System developers often require quantitative measures like throughput and response time to decide on design alternatives, on target system architectures, and later on for the optimization of parameters like timer settings, or window and buffer sizes.

It is notable that UML was not mentioned as a prominent language in used in protocol engineering. Although UML has become the de facto modelling standard it is not often employed in the protocol engineering process as a specification language. This is primarily because it is a general-purpose modelling language without formal semantics. As a work-around a common approach is to map a subset of UML diagrams to existing formal methods[BDM02, MC01, LQV01a] in order to allow automated analysis. An alternative approach is to merge UML with a formal language using a UML profile[Bjo02]. In addition to a lack of formal semantics, UML has not been used for modelling real-time systems (and in particular not used in the protocol engineering process) due to shortcomings in its architectural specification abilities[SR03a].

With the emerging UML 2.0 standard the Object Management Group (OMG) appears to have addressed the shortcomings of UML in the real-time modelling and protocol engineering domains. For example the architectural modelling capabilities of UML 2.0 has been drawn[Sel03] from both the ROOM modelling language[SGW94][SR03a] and SDL. With enhanced real-time architectural specification abilities and the semantic tightening[Sel04] using profiles, UML 2.0 appears poised to become the dominant specification language used in real-time modelling and protocol engineering.

A fair amount of research has been conducted on design and performance analysis of protocols using mature protocol engineering languages such as SDL[BMSK96][BMSK95][Ste98], Estelle and PROMELA. However this is not the case with UML 2.0[1]. Therefore in this dissertation we develop a methodology for the design and performance analysis of communication protocols using UML 2.0.

1.2 The proSPEX methodology and approach to performance evaluation

Our methodology involves the use of a subset of UML 2.0 diagrams to model architecture and protocol interactions. We investigate and place an emphasis on the design and performance analysis stages of the protocol engineering process. During the design stage, the engineer uses patterns for protocol system architecture[LTB98][PT00]. In our methodology the model is created in a *commercial* model editing tool, Telelogic Tau G2. Once the model has been verified a collaboration diagram depicting a simulation scenario is created by the user as a basis for defining system workloads and the properties of the target environment. The collaboration diagram is created in order to allow for simulation-based performance analysis.

In order to conduct simulation-based performance analysis we have developed the prototype proSPEX (protocol Software Performance Engineering using XMI) tool. proSPEX translates the UML 2.0 model into a simulation model that is capable of delivering a set of trace messages (as found in [BMSK96][BMSK95]) that allow for the calculation of a broad spectrum of performance statistics. In addition the tool uses the XMI-based *tool integration approach* that is advocated in the UML Profile for Schedulability, Performance and Time[Gro02] and used in a number of research projects[Ste03]. proSPEX also uses the

[1]Note that a significant amount of research has been conducted using UML 1.x involving general software performance engineering (see [BDM02, MC01, LQV01a]) however here we are interested in the more specific field of protocol engineering and its supporting FDTs.

JavaSim-based [Lit04] Simmcast[MB02] simulation framework at the core of the simulation models that are generated from UML 2.0 models specified using Telelogic Tau G2.

Our approach to protocol performance evaluation is influenced by the facilities provided by the simulation framework we have integrated into proSPEX and our associated methodology. In translating a scenario (specified using a UML 2.0 collaboration diagram) to a quantitatively assessable simulation model, the characteristics of the target environment is described by associating network resources to the specification. Environmental characteristics such as channel bandwidth and buffer space can be specified and realistic assumptions made regarding time and signal transfer. In the collaboration-based scenario the number of clients and servers that serve as the systems workload is indicated and network links are clearly identified with an attached UML 2.0 comment symbol. The comment symbol attached to network links contains network link characteristics such as loss probability, bandwidth, delay distribution and delay distribution parameters. The network link characteristics are used to schedule signal arrival at the receiver. For example, signals that cross network links must have a length parameter in order to allow for time calculations by proSPEX.

The simulation framework we have integrated into proSPEX allows for the modelling of *processing delay* by associating a send and receive time with a node (note the processing delay is not associated with individual actions). Therefore regardless of whether a signal crosses a network link or not, the node sending the packet and the node receiving the packet will be blocked for a simulated period of time equal to each node's *processing delay*.

We have specified the Efficient Short Remote Operations (ESRO) transport layer protocol and calculated throughput performance measures to prove the utility of proSPEX.

Our proSPEX methodology and approach to performance evaluation has clearly been developed in the context of related work. With regard to the use of a tool integration approach in which we interoperate with a commercial UML 2.0 tool, we note that in "*Performance Engineering of SDL/MSC Systems*"[MTMC99] A. Mitschele–Thiel and B. Muller-Clostermann made the following observation:

> Future goals should include **case studies** and **application of the approaches to real world problems**. This will help to develop a closer integration of performance engineering with the SDL methodology and implementation design. In order to systematically develop reliable and efficient systems, better and stable tools will be necessary that **interoperate with commercial SDL/MSC**

tools.

1.3 Project Aims

In this project our primary aim was to build a prototype tool that is able to translate from a UML 2.0 protocol specification to an equivalent simulation model capable of delivering a particular set[BMSK96] of trace messages by integrating existing tools[MHSZ96][MTMC99].

In addition, with UML 2.0 being a relatively new specification language, we consider the protocol design stage of the protocol engineering process which precedes both validation and performance analysis.

Our primary aim and supplementary aims are listed below. Note that our primary aim is to develop a methodology and performance analysis approach using UML 2.0. Our supplementary aims are centered on the theme of our use of tool integration in building proSPEX.

Primary aims:

1. To create a prototype tool capable of delivering a set of trace messages (as found in [BMSK96][BMSK95]) using a *tool integration approach*. The tool should translate from a UML 2.0 specification to a quantitatively assessable simulation model in which the characteristics of the target environment a represented using facilities provided by the simulation library (or tool) used.

Supplementary aims:

1. To design a communication protocol using Telelogic Tau G2, our model editor, as a case-study. It would be beneficial to identify communication protocol design patterns to be used in the protocol design stage, which precedes the performance modelling stage.

2. To build proSPEX following the XMI-based tool integration approach that is advocated in the UML Profile for Schedulability, Performance and Time[Gro02] and used in a number of research projects[Ste03]. Therefore proSPEX should be a model processor and not a model editor and have filters to Telelogic Tau G2.

3. To evaluate simulation libraries (such as JavaSim, SimJava or opNET) and use the most suitable library when translating from a UML 2.0 model to a simulation model.

4. To determine how a UML 2.0 model could be represented using the chosen library and adapt the library to include features that are required for our primary objective.

5. To determine means of specifying protocol environmental constraints and workloads in UML 2.0 and extend the protocol specification previously conducted to include such information.

6. To use the *tier generation model* described in [Her03] in proSPEX and hence use a text templating engine in the simulation code generation process.

7. To prove the utility of proSPEX delivering, as an example, a subset of the possible performance statistics derived from the simulation of our case-study communication protocol.

1.4 Project Assumptions and Limitations

A significant limitation is this project was that UML 2.0 was not fully defined for the duration of the project. That is, UML 2.0 existed only in language specification documents and a single editor (Telelogic Tau Generation 2.1) which is based on *draft* versions of the UML 2.0 specification documents throughout the duration of the project. Telelogic Tau Generation 2.1 combines SDL with UML 2.0 as examined in Section 3.4. We call the specification language used in Tau G2.1 *Telelogic UML 2.0* since it is a merger of SDL and UML 2.0 and *not a standard*.

An additional limitation was that Telelogic Tau G2.1 stores the model in an XML format and the DTD associated with this XML is a company internal and hence was not available for our use. This is a limitation since we were unable to determine the legal set of elements that may appear in the XML as well as in which context the elements may legally occcur. Ideally Telelogic Tau G2.1 should have used the XMI 2.0 format for if this were case the proSPEX user could use any UML 2.0 editor to create the model.

1.5 Own Contribution

The performance analysis of protocol systems using specification languages such as SDL has been thoroughly researched [MTMC99][BMSK96] [Rou01][Ste98][MDMC96] [MHSZ96][Spi97] [Mal99] in the last decade. With the emergence of UML 2.0 and its integration with SDL, as has been done by Telelogic, various questions arise regarding protocol performance modelling and analysis using the resultant language.

Questions that arise include the manner in which we can apply the techniques that were used in the case of SDL to this integrated language? In addition, when using such an integrated language various profiles and approaches to performance analysis using the UML language become applicable. An example of such a profile is the UML profile for schedulability, performance and time [Gro02], while and example of an approach is that which is detailed in *Performance Solutions* [SW02] by Smith and Williams. An additional question would be what the roles of the various UML 2.0 diagrams would be in the performance modelling process.

Our contribution is that we have created a model processing tool, proSPEX, and an associated methodology, which draws from the various sources that are related to the integrated modelling language we are using. For example, we have followed the recommendation [MTMC99][Gro02] for performance analysis tools to interoperate with commercial tools. In addition, our contribution is that in building proSPEX we have integrated Telelogic Tau and an extendable simulation framework. This is similar to the SDL/OPNET[2] approach [MHSZ96] which entails a mapping from SDL descriptions (annotated with constructs for describing delays, processing resources and workloads) to executable OPNET models. However, in the case of proSPEX the mapping is automated while with the SDL/OPNET approach the mapping is *manual*.

1.6 Dissertation Outline

Chapter 2, provides an introduction to the major languages used in network protocol specification, Estelle, SDL, PROMELA and UML. We then investigate the interpretation of time in the dynamic semantics of SDL, followed by an overview of approaches to general SDL performance modelling issues. Lastly we review SDL performance analysis tools including

[2]OPNET is a network simulation environment that that is discussed in Chapter 4.5.

SPECS, ObjectGEODE, SPEET, QUEST and SDL*.

In **Chapter 3** performance engineering with UML is discussed. We firstly investigate the known shortcomings of UML that are relevant in conducting performance analysis. We then review approaches to mapping to UML models, which are not necessarily protocol models, to performance models. An approach to the formalization of the new UML 2.0 standard, in order to allow for automated analysis, is then reviewed. We then investigate the UML-RT profile (the UML Profile for Schedulability, Performance and Time) and the role of XMI in the this profile.

With the analysis technique used in this dissertation being simulation, **Chapter 4** examines the topic of simulation to predict a communication system's performance. We also determine a set of requirements for the network simulation package used in this work and evaluate a set of commercial and open-source packages. The result of this evaluation is our choice of using the open-source Simmcast network simulation framework.

In **Chapter 5** the proSPEX methodology, semantic time model and high-level architecture are discussed, while implementation aspects are discussed in **Chapter 6**. With the proSPEX methodology a minimal subset of UML 2.0 diagrams are used to specify protocol architecture, behaviour and environmental characteristics. The purpose of our methodology is to serve as a guide of how to go about protocol design and in particular the specification of non-functional delay constraints and subsequent performance analysis. With the proSPEX tool we extend the Simmcast simulation framework with abstractions that are required to map from protocols specified using Telelogic Tau to an executable simulation model. Such abstractions include those used in SDL behavioural descriptions namely finite state machines, implicit and explicit addressing expressions. We also discuss the challenges we encountered in filtering the verbose Tau XML in the code generation process.

A performance analysis case study is discussed in **Chapter 7**. We demonstrate the utility of proSPEX by determining a set of performance statistics relevant to the use of the Efficient Short Remote Operations (ESRO) protocol in a wireless e-commerce scenario. Finally **Chapter 8** presents conclusions and suggestions for future work.

We also provide a set of Appendices with supplementary information. An introduction to UML 2.0 is provided in **Appendix A** and we discuss design patterns for communication system architecture in **Appendix B**. An overview of the ESRO protocol is given in **Appendix C**, while the proSPEX templates, which are used for simulation code generation, are provided in **Appendix D**.

Chapter 2

Protocol Performance Engineering with Formal Description Techniques

2.1 Introduction

In this chapter we review the major languages used in network protocol specification, Estelle, SDL, PROMELA and UML. These languages are used in designing the structure and dynamic behaviour of efficient and unambiguous communication protocols. Importantly Estelle, SDL and PROMELA are *formal* description techniques for unambiguous system specification, validation, verification, functional testing, rapid prototyping and performance analysis [BD02]. Extended communicating finite state machines (ECFSM) form the basis of protocol specification models in the languages we examine. The finite state machines are *extended* by having local variables and data and concurrently communicating using signals or structured messages sent via finite-length asynchronous channels[BD02].

The aim of this chapter is to provide a brief[1] overview of each language in the context of network protocol specification. In the case of SDL, the most widely used FDT for protocol specification, we investigate the semantics of time which is particularly relevant when conducting performance analysis. We then discuss the approaches to solving general problems encountered when conducting performance modelling using SDL. Finally, we discuss related

[1]Note that a comprehensive overview of formal methods for the specification and analysis of communication protocols by F. Babich and L. Deotto is available[BD02].

tools that integrate performance analysis in the context of SDL.

2.2 Estelle

The formal description technique Estelle is used for the unambiguous specification of communication protocols. Estelle is based on communicating extended finite state machines and has semantics which is both formal, mathematical and implementation-independent. Estelle is said[SBD89] to be particularly well suited to describing the services and protocols of the layers of the ISO Open System Interconnection (OSI) model.

With Estelle the Pascal programming language is used for data manipulation and the ECFSM model is used for behavioural description. Estelle can be seen as set of extensions to ISO Pascal in which systems are specified as a hierarchical structure of FSMs which run concurrently and communicate by message exchange and/or variable sharing[SBD89]. Notably the use of Pascal makes Estelle implementation oriented and hence eases implementation generation. Modular, interface-oriented design can be employed when using Estelle in that the communication interfaces between system components can be described separately from the internal behaviour of the components. For brevity we do not examine the Estelle building blocks and syntax here and for such information the reader may consult [CS90][SBD89] and [BD02].

Estelle has not enjoyed the commercial success of SDL or UML, however it is supported by the Estelle Development Toolset (EDT) and the language has been used in the specification[ea00] of the U.S. Army wireless standard MIL-STD 188-220. EDT was developed at the Institut National des Telecommunications (INT) and consists of a compiler, C code generator, simulator/debugger, MSC trace generator, test driver generator and graphical editor [TFKRB98] amongst other components.

2.3 SDL

SDL (Specification and Description Language), the ITU-T standard, has been used since 1976 and as such is mature and well tested. In this section we give a broad overview of SDL. For more detailed coverage of SDL consult [Con00][BD02].

SDL is both an object-oriented and formal language. It is primarily intended to be used to specify complex, real-time applications. Such complex applications would involve many

concurrent processes that communicate using discrete signals[Con00]. It is hierarchical in nature allowing information hiding and abstraction. An SDL specification can be graphical (SDL/GR) or in a textual phrase representation (SDL/PR). SDL/PR can be seen as being a high level programming language.

The following list[Con00][2] summarizes specialized characteristics of SDL:

- **standard** - SDL was developed and standardised by the ITU-T and it is accepted by the ISO. This means that SDL will be maintained and supported in the future.
- **formal** - Being a formal language ensures that SDL specifications have the essential properties needed in mission-critical applications, namely precision, consistency and clarity in design. The formal SDL grammar ensures that tools for both the simulation and validation of formal characteristics can be created.
- **object-oriented** - Apart from introducing object-oriented (OO) concepts for objects with behaviour, e.g. systems, state machines, SDL includes traditional OO features such as encapsulation and polymorphism.
- **highly testable** - SDL has formalisms for parallelism, interfaces, communication and time. The result is that SDL has a high degree of testability.
- **portable, scalable, and open** - An SDL specification is not dependent on operating systems, processors, interprocess communication mechanisms or distribution methods. Thus a SDL specification can be mapped to different target architectures and configurations.
- **highly reusable** - SDL provides a high degree of reuse. This is as a result of visual clarity, testability, OO concept usage, clear interfaces and abstraction mechanisms.

Compared to Estelle and PROMELA, SDL has enjoyed the greatest success in industry. The Tau tool, developed by Telelogic, has been used in industry for a number of years. Due to market demands Tau is no longer a *pure* SDL tool and is now based on a merger of UML and SDL in Tau Generation 2. At a recent[Mei02] ITU workshop on the use of description techniques it was noted that in the teaching field UML is not yet able to replace SDL, MSC, ASN.1 and TTCN. However despite advantages offered by these languages teachers may turn to UML due to readily available support in the form of books and tools.

[2]This list is largely an excerpt from [Con00]

2.4 PROMELA

PROMELA (PROcess MEta LAnguage) is the specification language of the Spin (Simple Promela INterpreter) tool that can be used for the formal verification of distributed software systems. Spin can be used to detect logical design errors in the specification of a variety of systems and in particular data communication protocols. Spin is not restricted to being used as model-checking tool and can be used as a simulator, an exhaustive verifier and proof approximation system.

PROMELA was first released in 1991 by G. Holzmann[Hol91]. It allows for the dynamic creation of a finite number of concurrent processes. Communication occurs via message channels that can be either synchronous or asynchronous. The language largely resembles C and is used in the specification of finite-state systems. A PROMELA model consists of type declarations, channel declarations, global variable declarations and an initialization process. The statements in a process are either executable or blocked depending on the type of the statement.

The XSpin tool serves as an interface to Spin and has a PROMELA syntax checker. Once a protocol has been specified using XSpin, system properties that can be checked included deadlocks, assertions, unreachable code, LTL formulae (propositional logic and temporal operators) and liveness properties. The default optimisation and reduction algorithms used to make verification runs more efficient are the root of Spin's power and hence its popularity[Ruy02].

Spin is freely available online[spi04] and enjoys active use in the research community. PROMELA is arguably the language of choice when conducting model checking in the protocol engineering field, however such model checking is rarely conducted in industry and mostly in academic research.

2.5 UML

UML, as the *de facto* modelling standard, has received significant attention in the field of real-time development. Although it is beyond our scope to detail the language, we provide a brief overview of the use of UML as a language in the protocol engineering domain.

With the creation of UML the intention was not to have a language with a programming language level syntax and formal semantics. The lack of the above mentioned features made UML unsuitable for code generation, model checking and performance analysis. As a

work-around UML tool vendors and academics have used UML *profiles* in order to provide for the needs of protocol engineering. Such profiles include the Graphical Protocol Description Language (GPDL)[JPT00], the ITU-T Z.109 Profile[MP00], the Rational Real-Time Profile[SR03b] (developed by B. Selic and J. Rumbaugh and used in the Rational RealTime tool) and the Telelogic Tau Real-Time Profile (used in the Telelogic Tau tool, see Section 3.4). In addition in academic research a common approach is to map a subset of UML diagrams to existing formal methods[BDM02, MC01, LQV01a] in order to allow automated analysis.

The tool support and use of UML in real-time development is extensive. In addition numerous books and papers are available to aid the developer. Real-time development tools include Telelogic Tau, I-Logix Rhapsody Developer and IBM Rational Rose RealTime.

2.6 Performance Analysis using SDL

2.6.1 Time in SDL-92

Performance analysis using SDL is only possible if one is able to determine the time taken for signal transfers and process execution. It well known [dVHVZ96] that the definition of time in the dynamic semantics of SDL is loose in the sense that it acknowledges that the system will execute in real time with delays on channels, but does not specify how the system execution is affected by this constraint.

In [BMSK96] (SPECS) it is noted that SDL-92 has an unsatisfactory interpretation of time. The following interpretation of time in SDL-92 is noted in [BMSK96]:

- time is incremented by a clock outside the system.
- no units of time are predefined i.e. time may be continuous.
- signal transfers over channels take time.
- an SDL system is not limited by processing resources. This implies that processes may perform SDL actions in zero time (or negligible time compared to the duration of a signal transfer).

The above interpretation of time unsatisfactory in the following ways:

1. ITU recommendations do not specify how the duration of delays introduced by channels is determined.

2. Channels are not perfect, as is assumed in SDL, and may lose signals.

3. One must drop the assumption that individual actions take negligible time or happen instantaneously.

The unsatisfactory interpretation of time in SDL-92 is addressed in [BMSK96] by making the following assumptions when attaching semantics of time to SDL:

- signal transfers over channels take a user-specified time, but transfers over signalroutes are not delayed.
- processes are only allowed to execute a finite number of actions in a time unit i.e. actions take time.
- different processes may execute a different number of actions per time unit i.e. processes can have different execution speeds.

2.6.2 Approaches to General SDL Performance Modelling Issues

In this Section we discuss *general* issues that arise when conducting performance analysis based on formal specifications and SDL in particular. We survey the performance analysis approaches taken in related tools that integrate performance evaluation into the context of SDL in Section 2.6.3 to 2.6.10 with regard to the general issues that are discussed here.

Modelling Non-Functional Duration Constraints

In the approaches that we survey, it is common for the semantic time model of SDL to be enhanced by providing means of modelling non-functional time dependant aspects. Semantic time models are realised by temporal features that are needed for functional design and also by time related features that are needed for non-functional aspects and analysis.

Time related features required for functional design include clocks, timeouts and time dependant enabling conditions. Time related features required for non-functional design include timing restrictions due to knowledge of the execution environment and modelling the execution times of tasks.

The means of modelling non-functional temporal aspects are missing from SDL [Gra02][Spi97], as is stated by Graf [Gra02]:

Non-functional primitives express timing features orthogonal to the functional behaviour, and they consist in constraints on the (relative) occurrence time of events, and are completely lacking in the standard.

It is this lack in the standard which is the subject addressed by the various approaches that we have mentioned. Each approach provides a means of modelling duration contraints that allow for the expression of timing characteristics of the environment and underlying execution system [Gra02]. Non-functional time related aspects include [3] [Gra02]:

- **Communication delays**: All communication in SDL occur via channels which may have an associated delay. Channel attributes may include a loss rate and whether the delay is load dependent or not. A communication channel with parallelism, such as the Internet, may be regarded as load independent, while a sequential medium would be load dependent.

- **Processing times**: The processing of a signal can be divided into queueing and *treatment* [Gra02] phases. The treatment time consists of pure execution and blocking time (due to scheduling). The overall processing time can be modelled as an expression representing a time interval. With SDL an important question that arises is for which sort of behaviours duration constraints can be specified. For example are durations constraints associated with SDL behavioral primitives (i.e. tasks, output, input etc.), SDL behaviour sequences (i.e. transitions or procedures), or SDL processes.

- **Execution modes**: With execution modes we consider time passage in parts of the system with no time constraints expressed. With standard SDL semantics time passage is interpreted as passing arbitrarily in such parts. A designer could specify a different execution mode, for example all non time constrained actions could be immediate.

- **Time constraints on the external environment**: The timing constraints of signals arriving from the environment must be expressible. Such characteristics include response time, inter arrival times and jitter. The environment can be modelled by processes in which the above mentioned signal characteristics can be expressed using time guards.

[3] We borrow from work [Gra02] by Susanne Graf in the list of non-functional time related aspects that are discussed.

- **Scheduling**: In order to represent scheduling algorithms in SDL, information regarding the preemptability of atomic steps, or sequences of atomic steps, must be provided. Questions of how or whether scheduling information should be represented in SDL is answered to varying degrees by the different approaches.

- **Local time**: the ability to express local clock time *and* global system clock time (the external reference time, or ***now***) is important in model-checking in order to detect unforseen errors such as livelock and deadlock. Moreover, the relationship between local time and the reference time must be clearly defined.

SDL Syntax

The second issue that we consider is whether the SDL syntax of the SDL specification is amended when using a particular approach. This is not independent of the first since the various non-functional time related aspects need to be represented to allow for automated analysis.

The importance of this consideration is that if one wants a tool based on an approach to be useful to the largest possible audience one would want to take existing SDL specifications and analyse them using the tool *without* having to change the given specification. In the context of this issue we examine the ways of temporal directives (e.g. delay and scheduling directives) to communication protocol specifications.

2.6.3 SPECS

With the SPECS [BMSK96][BMSK95] and SPECS II [dVHVZ96] tools, developed by Pieter Kritzinger and his colleagues at the University of Cape Town, SDL/PR specifications are imported and then executed. With both tools the approach to performance analysis remains the same in that it does not affect the syntax of the formal description technique and does not depend on the FDT used. SPECS performs a simulation of a system specified in SDL in order to derive performance measures. The described *SPECS approach* is essentially a way of mapping a system specification to its *target environment* (without changing the syntax of the FDT). In this mapping realistic assumptions are made regarding time, process execution and signal transfer. In their approach a protocol system, specified using standard SDL, is imported and then attributed with environmental constraints.

Once a specification has been imported relative execution speed values are assigned to

each block while the processes within a block are given weights. The assignment of these values, which is done using a GUI dialog box, is equivalent to annotating the model using comment symbols.

The units of the execution speeds are *actions per time unit* meaning that the number of actions each process can execute (the process *action quota*) once scheduled is determined by its weight. In this way time, which is maintained by a global simulation clock, is advanced either when process instances have exhausted their action quotas or process instances are all waiting for input. At each advancement of the simulation clock, timers, which are maintained separately by each process instance, are checked for expiration.

With regards to the semantics of time when considering signal transmission, SPECS enhances the standard semantics of SDL by assuming that signal transfers over channels experience a randomly distributed delay and may be lost with a certain probability.

2.6.4 ObjectGEODE

In "SDL Performance Analysis with ObjectGEODE"[Rou01] J.-L. Roux describes the performance analyzer of the *ObjectGEODE* simulator. In the ObjectGEODE performance evaluation approach performance analysis directives can be directly attached to low-level behaviour such as actions. SDL extensions for performance analysis are found in SDL comment strings in order to avoid modifying the standard SDL syntax. *Node*, *priority* and *delay* directives are placed in comment strings and used by the analyzer. Priorities can be assigned to each SDL process in order to specify execution scheduling preference using priority directives associated with processes. Delay can be associated with individual actions rather than transitions using the delay directive. The delay directive contains a distribution parameter which is uniform by default but which may also be exponential and normal. During simulation measurements are not stored (to file) but processed immediately to save memory usage and preserve the efficiency of the simulator. It is important to note that with ObjectGEODE to the extensions to the SDL language are used *exclusively* in SDL comment symbols and hence the models annotated models still conform to the Z.100 standard.

2.6.5 SPEET

With SPEET [Ste98], by M. Steppler and M. Lott, Aachen University of Technology, the processing delay is modelled by emulating the target environment and not by using delay

directives as is done in other approaches. In other words a system's implementation can be simulated on *virtual hardware*. Users can specify parameters such as processor number, processor frequency and RAM parameters (amount, type and cycle times). In the case of communication delay, physical transmission models (channel encoders, decoders, modulators and demodulators) are used.

MSCs with time constraints serve as the workload of a protocol specification. These constraints are set using comment symbols in MSC diagrams and a particular syntax. In addition, the user can insert *probes* into the SDL specification which do not interfere with the semantics of the specification. These probes can be connected to any SDL symbol to which an SDL comment symbol can be attached. Time probes have a name, id and time expression as parameters. In effect these probes merely specify values that are traced and hence of interest in the analysis which follows simulation runs.

2.6.6 QUEST

With the QUEST [MDMC96] approach, developed by Bruno Muller-Clostermann and his colleagues at the University of Essen, the SDL language is extended, resulting in the QSDL (Queueing SDL) language. QUEST implements an approach to SDL-based performance evaluation aimed at obtaining estimates of QoS paramaters during the early design stages. A quantitatively assessable model is constructed enclosing the complete SDL-specification during which the characteristics of the target platforms and environment are described by associating resources (or machines) to the SDL-specification. Environmental characteristics such as processing power, available channel bandwidth and buffer space are specified by the user, in addition to traffic and workload models. The SDL-specification with integrated environmental property specification yields a quantitatively assessable model using state space exploration via discrete event simulation or exhaustive exploration. System performance is determined in terms of throughput, response time and utilization.

QSDL has a QSDL/GR notation which has equivalent diagrams for each SDL/GR symbol. An SDL process is a *machine* (queueing station) in QSDL and the parameters that can be associated with a *machine* includes a name, server number, service discipline (e.g. FCFS, RANDOM), a set of offered services and service-specific speed values. Each QSDL *request* instruction is time consuming and requires a service amount attribute and an optional priority. In this way time durations and the use of resources can be associated with *certain* actions. For workload characterisation, a number of random distribution functions

are provided. These functions would be used in load generators which are implemented as QSDL processes.

2.6.7 SDL/OPNET

In [MHSZ96] Martins et. al. introduce *Extended SDL* of which SDL-92 is a subset. Processing delay is modelled using delay clauses associated with transitions. All actions before such a clause are executed immediately, following which execution is suspended for the specified delay period. Unreliable communication links can be modelled explicitly by using a channel substructure.

With the *Extended SDL* processing model, the FIFO queue(s) associated with a process can be specified as being bounded, thereby modelling buffer size. In addition, each queue has an associated queueing discipline. The process is viewed as a server which is characterized by its service durations which are specified using delay clauses in transitions. Workload is specified using processes interacting with the system via environmental interaction channels.

Each formally specified Extended Finite State Machine (EFSM) is mapped to a combination of a EFSM model and a queueing network. The queueing network describes the congestion of multiple requests to restricted resources. *Extended SDL* models are manually mapped to OPtimized Network Engineering Tool (OPNET) models and then executed.

2.6.8 SDL*

SDL* [Spi97] is an annotated version of SDL that allows for the specification of non-functional aspects. Such constraints include resource requirements, timing constaints and hardware mapping proposals. The SDL* annotations are embedded in SDL comment symbols, thereby allowing for the use of commercial for specification and validation. SDL* annotations are classified into five classes namely tool directives, mapping, resource requirements and cost requirements classes. The main contribution of SDL* is to provide a syntax for the specification of non-functional timing aspects using SDL using comment symbols.

As an example of the specification of a resource requirements using SDL* annotations, channel bandwidth could be specified using the following directive: **bandwidth of channel** c1 **is** 200 **Mbit/sec**. With regard to timing requirements SDL* gives the user the ability to specify both signal processing delay and jitter duration directives.

2.6.9 Timed SDL

With Timed SDL (TSDL), by F. Bause and P. Buchholz, timing aspects are added to transitions. Every transition of a process has a time delay or probability specified and modifications to the SDL syntax are needed for this purpose. Modifications to SDL's syntax include constructs for expressing transition execution probability, transition execution rates and transition time durations. Additional TSDL constructs are provided for inspecting the input queue and state of a process. The motivation for such constructs is to enable the evaluation of performance metrics such as protocol throughput by determing the signal flow out a state.

2.6.10 PerfSDL

PerfSDL, an extension to SDL, serves as an interface to the PlasmaSIM network simulator. In [Mal99] the inadequate semantics of time found in SDL is addressed in the context of simulation. Instead of time advancing when all queues of the system are empty, as occurs in the SDL standard, time can advance at every action performed by an SDL process. In order to reduce the system complexity, timed-transitions can be specified. Actions, and not state transitions, are the atomic units of scheduling and are called *sub-transitions*.

An execution time period, probabalistic or deterministic, may be associated with each *sub-transition*. *Probes*, which are special variables used for statistic gathering, are associated with transitions. Probes can be used for logging variable values and can be time-stamped. These probes are essentially discrete simulation trace events.

2.7 Summary

In this section we have briefly examined Estelle, SDL, PROMELA and UML. In terms of language use UML appears to be the most prominent language in real-time development, this prominence is spurred on by readily available teaching material and tools. We examined the semantics of time as defined in SDL as well as the deficiencies in these semantics when one is conducting performance evaluation. In our examination of related tools, we saw that each approach enhances the semantic time model of SDL and that this process entails providing means of modelling non-functional duration constraints.

Chapter 3

Performance Engineering with UML

3.1 Introduction

In this chapter we examine the use of UML[1] in the performance engineering field in general terms. That is, we examine the use of UML when conducting the performance analysis of any type of software, not necessarily communication software, where performance is of particular importance. Our intention is not to examine the performance analysis techniques used, rather the approach taken in the use of UML language diagrams and features in specifying the abstractions required for performance analysis.

We investigate some of the shortcomings of UML 1.x, followed by a survey of the abstractions used in mapping from UML models to performance models, e.g. queueing models, simulation models and Petri-net models, regardless of the performance analysis technique used. We then investigate an approach to formalizing UML 2.0 for automated communication software analyis.

The final topic that we examine in this chapter are efforts the OMG in standardising the use of UML in the performance engineering field. The UML Profile[2] for Schedulability, Performance, and Time Specification (UML-RT), defines a standard paradigm for the use of UML when modelling the time, schedulability and performance aspects of real-time systems. We investigate the model processing approach advocated in UML-RT which involves use of

[1] *UML* means UML version 1.x (1.5 and below), not UML 2.0, throughout this chapter.
[2] The meaning of a UML *profile* is examined in the Appendices.

the XML Metadata Interchange (XMI) standard in Section 3.5.3.

3.2 UML 1.x Shortcomings

Here we mention various shortcomings, some of which are subjective, that have been noted in UML 1.x and spurred on the creation of the UML 2.0 standard. Note that is is beyond the scope of this dissertation to detail UML 1.x or UML 2.0, although a brief introduction to UML 2.0 is provided in Appendix A.

Real-time Constraints and Properties

UML was conceived as a general-purpose modeling language and as such real-time modeling constructs were not included. There has been extensive research in this area[LQV01b][MC03] and the recent adoption of the UML profile for schedulability, performance and time (UML-RT) [Gro02] highlights the importance of real-time modeling using UML. UML-RT was however developed with UML 1.x in mind and has not been upgraded for UML 2.0.

Time

According to Graf and Ober[GO03] standard UML 1.x has the *Time* data type as its only time related concept. In addition, [GO03] points out that while UML-RT does define a vocabulary of time related concepts, these concepts are largely syntactical.

Several temporal concepts have been added to UML, as can be found in the UML 2.0 Superstructure Specification [Gro03]. With UML 2.0 being standardized relatively recently (June 2003) no UML 2.0 tool incorporating these features currently exist.

Diagram Exchange Industrial Implementation

In an effort to allow model sharing between tools supplied by different vendors, the OMG adopted the XML Model Interchange (XMI) language. Certain UML modelling tools, such as Poseidon, save UML models using XMI as their native format while others, such a Rational Rose, have the ability to save and load XMI models as a foreign format.

Unfortunately tool vendors did not adopt a common interoperable version of XMI[Mil02] in their UML 1.x tool implementations. That is, incompatibilities between XMI written by different vendors exist and this is said [Ste03] to be partly due to shortcomings in the

standard itself. It is expected that with the XMI 2.0 standard, the diagram interchange format of UML 2.0, the OMG has addressed the short comings of the earlier XMI standard allowing for compatible implementatations by tool vendors.

Language Size

UML 1.x is seen by some[Kob02] as being too large and complex. This excessive size makes UML difficult to learn, apply and implement. Kobryn[Kob02] has pointed out that the best place to start reducing UML (in the UML 2.0 standard) is to define a concise and precise language kernel. Such a language kernel would be the 20% of the language that is used 80% of the time.

The UML language has grown in version 2.0[Sel04], however as a remedy UML 2.0 has been modularized into a set of sublanguages[Sel04]. In addition a language kernel (the *Kernel* package) is described in the UML 2.0 Superstructure Specification [Gro03] which must be implemented by tool vendors as a minimum requirement for UML 2.0 conformance. "The *Kernel* package represents the core modeling concepts of the UML, including classes, assiciations, and packages...[it] is the central part of the UML..."[Gro03].

Model Multiplicity

Dov Kori has stated[Dor02] that model multiplicity is one of the main problems of UML. Model multiplicity occurs as a result of excess diagram types and symbols. Furthermore, Kori states:

> "UML agglomerates nine diagram types, also called views, or models, declared to be a unified standard. But such a declaration cannot replace unification of the concepts and symbol sets associated with the models, along with removal of the many redundant entities and overlapping notions."[Dor02]

The crux of Kori's argument, being the model multiplicity problem, is that not one of the nine UML models clearly shows the two most prominent system aspects, namely structure and behaviour. He points out that separating a model's behavior from its structure severely hampers an architect's productivity. Additionally model multiplicity has the added problem of integrity maintenance among the system's various models (modifying one diagram means that several other may also have to be modified). Kori states:

> "even with superb CASE tools, keeping the diagrams synchronized and preventing the introduction of contradictions and mismatches in the overall system model (which, in UML, exist only in the modeler's mind) become daunting tasks beyond anyone's cognitive ability."[Dor02]

Although the model multiplicity problem would may still exist in UML 2.0, it can be said that architectural modelling capabilities of UML have been significantly enhanced in the UML 2.0 standard.

Non-standard Implementations

According to Kobryn[Kob02] no modelling tool vendor had fully implemented the UML 1.1 specification or any subsequent UML 1.x specification four years after the UML 1.1 standard was released. Defining a language kernel is said [Kob02] to be a possible remedy to this problem and such a kernel has been defined in the UML 2.0 standard.

3.3 Approaches to Mapping from UML models to Performance Models

In order to allow for performance analysis using UML a common approach is to map a subset of UML diagrams to performance models with formal semantics. In this section we survey work in which UML models are mapped to performance models.

In [CCS03] a mapping from UML 1.x to software tools which support Jane Hillston's Performance Evaluation Process Algebra (PEPA) is described. UML models are enhanced with performance information and then mapped to PEPA. In essence the mapping involves a translation (or bridge) between the ArgoUML modelling tool and the PEPA Workbench. The UML model is stored in XML Metadata Interchange (XMI) format and standard XML tools are used to extract the required model data. An extractor tool and a reflector tool act as the bridge between the two applications. The extractor takes the XMI file generated when a UML model is saved using ArgoUML and converts it to the corresponding input file for the PEPA Workbench. Detailed algorithms[3] for extracting PEPA models from UML 1.x state diagrams and collaboration diagrams that are used by the extractor are presented in [CCS03]. The reflector takes the original XMI file and the results returned from the PEPA

[3]These algorithms are shown to be surprisingly complex.

Workbench and returns a modified XMI file which is then imported into ArgoUML, thereby integrating performance analysis results with the original model.

Hoeben [Hoe00] has developed a tool that translates a UML model to a queueing network representation. A set of UML diagrams are used in order to obtain *resource usage* estimates. The diagrams used are use cases, to model the system workload, class diagrams, interaction diagrams (sequence and collaboration diagrams), component diagrams and deployment digrams. The focus of [Hoe00] is the presentation of a set of rules to deal with some of the abstractions (and hence missing information) which can be expected when modeling distributed systems with UML.

Cortellessa and Mirandola [CM00] have developed a means of deriving a queueing network based performance model from UML diagrams. Performance validation consists of two distinct steps, namely model generation and model evaluation. The model evaluation step is supported by classical solution techniques and tools that can be used by software analysts who are not experts in the solution technique. On the other hand, model generation, the step that precedes model evaluation, is not supported by step-by-step methodologies that allow software analysts to automatically derive a performance model. Cortellessa and Mirandola describe a methodological approach for a systematic and automatic generation of queueing network models from UML diagrams. In their methodology, Use Case diagrams are used to derive software scenarios, Sequence diagrams are used to derive an Execution Graph and a Deployment Diagram is used to derive an Extended Queueing Network Model.

Hopkins, Smith and King [RHK02] have mentioned two approaches to deriving performance models from UML design specifications. The first approach (called the *hard* approach) is to develop a mapping from UML to a standard formalism such as a process algebra, Petri net or queueing network. Once the translation mechanism is in place the software designer can *generate* performance engineering models. The second approach (the *soft* approach) involves cooperation between the software designer and performance analyst resulting in the generation of a set of UML design specifications using the UML notation. The important aspect in this approach is the use of UML as the communication medium between the two parties. In [RHK02] the soft approach is followed. This involves the development of Stochastic UML, which will be based on a *restricted* subset of UML with additional notation that will allow the performance analyst to develop models. They identify statechart diagrams, collaboration diagrams, activity diagrams and sequence diagrams as candidates for conveying performance modeling information. They concentrate on

collaboration diagrams with embedded statecharts and propose UML extensions, namely probabalistic choice and stochastic delay. They investigate how concepts described in the UML profile for schedulability, performance and time may be best applied to represent the extensions of stochastic delay and probabilistic choice.

Bernardi, Donatelli and Merseguer [BDM02] propose the use of automatic translation of Statecharts and Sequence Diagrams into Generalized Petri nets and the composition of the resulting net models. Both Sequence Diagrams and Petri nets are used to be able to asses the consistency between the two descriptions. The central contribution of their work is to establish relationships between Sequence Diagrams and Statecharts according to the UML metamodel. The performance evaluation approach in [BDM02] is composed of three steps; firstly the UML diagrams are extended with performance annotations by using tagged values, secondly the translation of the extended UML diagrams to labelled stochastic Petri net (SPN) models and thirdly a final composition of the modules into a single model representing the whole system behavior.

In [CLW02], by Lindemann and Thummler, extensions to state and activity diagrams are proposed that allow the association of events with exponentially distributed and deterministic delays. Their contribution is an algorithm for state space generation that allows quantitative analysis by means of the generalized semi-Markov process. Their tool contains filters to commercial UML design packages (Rose and Rhapsody), meaning that the diagrams can be imported into their tool and additional timing specifications added.

Lavazza, Quaroni and Ventulli [LQV01a] have taken the common approach of translating a UML specification into a formal notation. They translate UML specifications into TRIO (a first order temporal logic) using only state diagrams and to some extent class diagrams. The TRIO specification is then tested by a history checking tool which exploits the formality of TRIO. Their automatic translation tool takes as input the XMI file generated by a UML CASE tool, parses it and applies rules to generate TRIO axioms. The XMI input file must be edited in order to make it compliant with the UML extensions they have defined. The Generalised Railroad Crossing (GRC) problem was used to both test UML as a real-time specification language and as a test-bed to verify the viability of their proposed approach.

3.4 Formalizing UML 2.0 for Automated Communication Software Analyis

As we discussed in Section 2.5, UML 2.0 is not a formal language yet tool vendors catering for the telecommunication industry, notably Telelogic, work around the problem by applying the International Telecommunication Union Recommendation Z.109 "SDL Combined with UML" to UML 2.0. Z.109 is a UML profile meaning that it specializes UML using stereotypes, tagged values, constraints and notational elements. Z.109 defines a one-to-one mapping between a specialized subset of UML and SDL. In addition, "The intention with Z.109 is however not to use the specialised UML *instead* of SDL, but to use SDL *combined* with UML"[MP00], the implication being that not every concept in SDL has a mapping to UML.

The reason why Z.109 was created in the first place is due to the core differences between UML and SDL, and to be able to take advantage of the strengths of the respective languages. SDL offers the advantage of firstly being *more concise* and secondly it is a *complete language*. When we say SDL is more concise, we are referring to UML offering *several* views of the same system while SDL focuses on the Object and State Machine views of a system. When we say SDL is a complete language, we mean that it has complete semantics, including execution semantics for state machines, graphical/textual grammars and a syntax for the specification of actions. In contrast to SDL being a complete language, UML has weak semantics with may variation points.

Telelogic has created a Real-Time Profile for UML 2.0 [Dol03] that is implemented in their Tau Generation 2.1 tool. It is based on a combination of draft UML 2.0 standard specifications and ITU-T Z.109 (which is being updated to be consistent with UML 2.0).

The Telelogic Tau Generation 2.1 Real-Time Profile

Here we mention aspects of the Telelogic Tau Generation 2.1 Real-Time Profile (Tau-RT) in the broader context of real-time development using UML 2.0. It is firstly important to note that it is not a standard profile but rather a proprietary profile, unlike the UML Profile for Schedulability, Performance and Time Specification. In fact Tau Generation 2.1 has an associated non-standard "UML Textual Syntax" which "is based on SDL, but modified to be more easy to use for C++ and Java programmers"[Tel03]. This syntax is used in external files, text diagrams and symbols in diagrams. Since Tau supports model verification and

complete code generation it is expected that they have developed a "UML Textual Syntax" based on SDL since UML does not have such a required syntax, as mentioned in Sect. 3.2.

It is also noteworthy that competing real-time UML tool vendors, such as I-Logix and Rational have not taken the Telelogic approach of combining SDL and UML in a proprietary profile. B.P. Douglass, author of numerous books on real-time development with UML, has warned[Dou03] of the risks of using proprietary methods.

> A proprietary method locks the engineer into a single vendor, with no possibility of hiring staff experienced in that approach, no available reference books, training available only from that single vendor, and into a single tool which may, or may not, meet your needs.

As a final point regarding Tau-RT it is worth noting that Telelogic makes no mention of its level of UML 2.0 compliance as discussed in the UML 2.0 Superstructure Specification[Gro03]. Telelogic Tau does not support the XMI 2.0 diagram interchange standard and it is entirely possible that with the incorporation of the proprietary Tau-RT Profile compliance is minimal.

3.5 The UML Profile for Schedulability, Performance and Time Specification

The UML profile for schedulability, performance and time specification (UML-RT) was developed by a working consortium[4] comprising the major real-time tool vendors (Rational, Telelogic, I-Logix et. al.) in conjunction with the Object Management Group (OMG). Here we briefly introduce UML-RT by mentioning its primary intentions and the motivation for its creation. We discuss the core of UML-RT, resource modelling, in Section 3.5.1 and then performance modelling in Section 3.5.2.

With UML-RT, the **intention** is for the definition of *standard paradigms for the use of UML* when modelling the time, schedulability and performance related aspects of real-time systems. The benefit of these standard paradigms are[Gro02] that they:

- enable the construction of models that could be used to make quantitative predictions regarding these [schedulability, performance and time] characteristics,

[4]The consortium consulted with various real-time domain experts including Bruce Douglass (I-logix), Prof. Dorina Petriu (Carleton University), James Rumbaugh (Rational) and Prof. Murray Woodside (Carleton University).

- facilitate communication of design intent between developers in a standard way,
- enable inter-operability between analysis and design tools.

With UML-RT the **intention** is also for the analysis of *models* of software systems that are created *prior to a line of code being written*. Note that such a model is at a higher level of abstraction than a protocol specified using protocol engineering languages such as SDL and Estelle in which code is embedded in graphical notation.

Finally, with UML-RT a primary **intention** is to give modellers the ability to annotate a UML model in such a way that various analysis techniques will be able to take advantage of the provided features.

A major **motivation** for the creation of the UML-RT profile specification was the *lack of a quantifiable notion of* ***time*** *and* resources acting as an impediment to the use of UML in the real-time domain. The UML-RT working consortium found that UML had all the requisite mechanisms for addressing these issues by using UML profiles.

3.5.1 Resource Modelling

UML-RT offers a single unifying framework that captures the essential mechanisms used to derive temporal analysis models from application models, regardless of the specific analysis method being used. This framework has a so-called *general resource model* (a common model of resources and their QoS attributes) at its core.

The role of the general resource model is [Gro02] to:

- Specify patterns that are present in many real-time analysis methods.
- Remove ambiguity by defining a common terminology and conceptual framework for defining resource QoS characteristics.

3.5.2 Performance Modelling

The performance modelling section of UML-RT describes the performance analysis domain concepts and viewpoint taken, followed by a mapping of these concepts and viewpoint into UML equivalents. That is, UML stereotypes are derived from domain concepts. In this section we mention relevant aspects of the domain concepts followed by their representation in UML.

With UML-RT performance modelling occurs within a *performance context* which specifies scenarios which involve use of a specific set of resources. A performance context may be a period of time during which maximum processor load is expected. Each scenario within the context is divided into sequences of *steps*. These steps are at the finest level of granularity in the UML-RT domain viewpoint and represent an increment in the execution of a particular scenario. The level of granularity of a step depends on the abstraction layer chosen by the modeller, but it is accepted that the level of abstraction is higher than that taken when implementing the model using a programming language. Apart from a performance context with scenarios that are divided into steps, the domain model includes abstract resources, processing resources, passive resources, abstract workloads, open workloads and closed workloads. Each domain concept has a set of attributes which are mapped to UML. For example, attributes that can be associated with a step include host execution demand, delay, response time, execution probability, interval between repetition and number of repetitions.

In mapping the performance domain concepts to UML equivalents, the UML-RT specification notes that scenarios, which play a key role in the domain concepts, are directly modelled using a collaboration-based approach and activity-based approach in UML. In the collaboration-based approach scenarios map to interactions (which are specified using *sequence diagrams*). In the activity-based approach each scenario is captured using an *activity graph*. In both approaches processing resources are modelled using *deployment diagrams* or directly by associating stereotypes with parts of the diagrams depicting the scenarios.

UML-RT specifies UML extensions in the form of a set of stereotypes, each of which has a list of associated *tags*. Each instance in the sequence diagrams, activity graphs or deployment diagrams would be stereotyped using the stereotypes detailed in the specification. A *tag value language*, detailed in UML-RT, is used to specify the format and syntax used for the tags.

3.5.3 The Role of XMI in UML-RT

The XML Metadata Interchange (XMI) is a standardized format for exchanging UML models. XMI 1.x had shortcomings the most significant of which was that information regarding diagram layout could not be represented. With XMI 2.0, the diagram interchange standard of UML 2.0, the significant shortcomings of XMI 1.x have been addressed.

In the UML-RT specification model processing (the process by which a UML model

is analyzed by some model analysis technique) takes place in the context of a general model-processing framework. In this framework XMI is used as the bi-directional interface between the UML model editor and model processor. The modeler constructs a model in a UML editor and annotates it with performance information required by a particular model processor. The model processor converts the UML model to a domain model, analyzes the domain model and then returns the results of the analysis by annotating the original model. The most important aspect of the UML-RT model processing framework is that the process is automated and all the details of the model processing is hidden from the user. A detailed example of the mentioned approach is given in [CCS03].

3.6 Summary

In this chapter our aim was to gain practical insight into how QoS attributes could be specified using UML 2.0 and to gain technical insight into how one could map from a UML 2.0 model to a simulation model. In other words we were concerned with the UML model annotation and the UML model to performance model translation processes. We have also investigated the UML-RT profile and make observations regarding the use of this profile in the context of the aims of this dissertation. Our key obervations are categorised below.

UML Model Diagrams and QoS Annotations

From our investigation in Section 3.3 no common set of UML diagrams or QoS annotations used as the basis for performance analysis emerged. We see this as being a result of UML serving as an interface to different performance analysis models and related tools in most cases. The lack of uniformity in the QoS annotations and diagrams that we found in Section 3.3 is one of the aspects the UML-RT specification aims to address.

UML Model to Performance Model Conversion

In Section 3.3 we investigated approaches to mapping from UML models to performance models. Several different methodologies for transforming UML models to performance models emerged and of these the most common approach was found to be the use XMI as the interchange format. The work done in [CCS03] was found to closely resemble the model-processing framework of the UML-RT specification in that both an *extractor* (a UML model

to PEPA model converter) and *reflector* (which integrates the analysis results with the original UML model) are built in to the bridge between ArgoUML and the PEPA Workbench.

UML-RT Abstraction Layer

With UML-RT the intention was to analyse models of software systems that are created prior to the implementation (or coding) stage of the development process. In this dissertation we use UML 2.0 combined with SDL, and not UML 1.x which the UML-RT profile caters for. UML-RT thus caters for modelling that is at a higher level of abstraction than protocol engineering languages (Estelle, SDL and UML 2.0 extended with SDL). The UML-RT profile is currently being upgraded to align it with UML 2.0 and as such using the UML-RT profile once this alignment is complete would be appropriate. What the UML-RT profile does provide however is insight into how we could annotate our models using UML 2.0 combined with SDL as well as a model processing framework which ensures that performance modelling is integrated within the design process.

Chapter 4

Simulation to Predict Communication System Performance

4.1 Introduction

The performance evaluation of a communication system is a process involving specification, analysis and optimization. In this chapter we focus on performance evaluation using simulation models. More precisely, we examine the use of process-based discrete event simulation when modelling communication system behaviour using state machines. We discuss the application of simulation for our purpose and do not refer to aspects of parameterization or workload characterization. We firstly motivate the use of simulation as a modelling technique in our context (communication protocol performance modelling and analysis), rather than analytic modelling, in Section 4.2. We then discuss the use of process-based discrete event simulation models when modelling the execution of protocols specified using protocol specification languages in Section 4.3. We discuss the importance of model verification and validation in the context of simulation in Section 4.4. Finally, we detail our simulation package requirements in Section 4.5.1, consider candidate packages (CSIM, OPNET, OMNeT++, NS-2 and Simmcast) and select the Simmcast simulation framework for use in the construction of our tool.

4.2 Motivation

In contrast to analytic models, such as queueing or markovian models, simulation models are algorithmic abstractions that represent the behaviour of the system when executed. Some [Hil01] cases[1] in which simulation is preferable to analytic modelling are listed below and discussed.

- When **analytic abstractions and assumptions** are **not appropriate**.
- When the **transient behaviour** of a system is **important**.
- When the **size of the state space** using an **analytic model** is **too large to be stored**.

The first situation in which simulation is preferred is when the level of abstraction and assumptions in analytic models is not appropriate, such as when using Markovian modelling, in which inter-event times are exponentially distributed. With simulation a system can be modelled at a greater level of detail, bearing in mind that there are performance penalties associated with simulation runs when using elaborate models [BCNN01][Inc98].

Simulation is also preferable when the *transient* period, which occurs prior to the system exhibiting regular or *steady state* behaviour, is of interest. Analytic solutions, which generally capture the behaviour after steady state has been reached[2], are not appropriate for systems where either a steady state is never reached or transient behaviour is of interest. In the case of communication protocols, the transient behaviour is of interest since resources, such a buffer space, may be exhausted prior to reaching a steady state.

Another situation in which simulation is prefered is when constructing and storing the complete state space of the model, as is required in most cases when solving models analytically, is not possible. With simulation models, the state space is explored during execution and thus the entire state space is not stored.

[1] A comprehensive discussion of when simulation is appropriate can be found in [BCNN01].

[2] In some cases transient solutions are possible analytically

4.3 Using Process-Based Discrete Event Simulation to Model Protocol Execution

In our investigation of formal description techniques (such as Estelle, SDL and PROMELA) in Chapter 2, we saw that process (or active class) behaviour is commonly specified using extended asynchronous state machines communicating using messages. A system described with communicating state machines can be readily modelled using process-based discrete event simulation.

With discrete event simulation, the state variables change at instants in time and the system is only considered at these [Ari]. In the case of communication protocols the discrete events tend to be signal arrival and departure. A signal generally either contains protocol data (e.g. protocol or service data units) or timer data (e.g. a time-out) as parameters.

When one considers process-based discrete event simulation, as opposed to discrete-event simulation, the future event list (FEL) of the simulation scheduler contains processes. The order in the FEL is determined by the time of the next event in the event sequences of the individual processes. Each process has an independent thread of control and generates *event notices* (the event type and time). The simulation clock procedure, which controls the simulation by traversing the event list, finds the event notice with the smallest time, updates the simulation time and gives the process which generated the event the right to execute, resulting in pseudo-parallel execution.

Simulation programs are generally written using any one of a number of simulation packages. When using a simulation package, the implementation of the process-interaction is generally hidden from the modeller's view. That is, processes scheduled on the FEL and being placed on the FEL whenever they face delays, causing one process to temporarily suspend its execution while another process proceeds, is not visible to the user [BCNN01]. The modeller using a simulation package *must* [BCNN01] have both a basic understanding of the concepts and a detailed understanding of the hidden rules of operation of the simulation package.

4.4 Model Validation and Verification

Regardless of the analysis technique, the performance metrics extracted from a performance model will only be useful if the model is a *good* representation of the real system. While

the notion *goodness* is subjective, from the performance modelling perspective we define it to be the degree to which performance measures extracted from the model match those of the real system.

Modelling by definition involves some form of abstraction in order to make analysis both tractable and efficient. This abstraction process is inevitably coupled with a set of assumptions. Model validation [Hil01] is the step used for judging how good a model is with respect to the real system.

1. Verification: determine whether the model works correctly regardless of whether it represents the real system.

2. Validation: determine how well the model represents the real system.

In this work discrete event simulation is used as the analysis technique. Established techniques, some of which are specific to simulation modelling, can be used for model verification. These techniques[3] are:

- **Antibugging**: this is a technique in which additional checks and outputs are included in the model to check that the model behaves as expected.

- **Structured walk-throughs**: the modeller explains the model to other people and in the process may focus on aspects of the model that can lead to the discovery of implementation problems.

- **Simplified models**: the model is reduced to its minimal possible behaviour and only made more complex once the simplified model is correct.

- **Deterministic models (simulation only)**: replacing random variables with deterministic values can help modellers to see whether the model is operating correctly.

- **Tracing (simulation only)**: trace outputs are useful in isolating incorrect model behaviour and is most useful in finding errors that have already been established by other[4] means.

- **Animation (simulation only)**: animation is a graphical form of tracing and hence is also most useful in finding the location of errors that have already been established by other means.

[3]Our list is sourced from and discussed comprehensively in [Hil01] by Jane Hillston
[4]For example program crashes or erroneous output.

- **Seed independence (simulation only)**: modelling errors can be uncovered when different random number generator seed values are shown to produce vastly different results.
- **Continuity testing**: this technique involves testing whether slightly different input parameters produce slightly different output values. Sudden changes in output values are often an indication of modelling error.
- **Degeneracy testing**: this technique involves checking for model errors for boundary system and workload parameters.
- **Consistency testing**: this technique involves checking that the model output is similar when the workload is kept the same but arranged differently. An example of a different arrangement would be to use two clients transmitting at half the rate of a single client.

Aspects that should be considered when validating a model include the assumptions, input parameter values (and distributions), output values and conclusions. Approaches to model validation include expert intuition, measurements using real systems and theoretical analysis (e.g. using analytic models, such as Markovian models).

4.5 Network Simulation Package Review

The use of simulation software forms a central part of this work, as we have stated in our aims (listed in Section 1.3). One our primary aims was to

> Create a prototype tool capable of delivering a set of trace messages (as found in [BMSK96][BMSK95]) using a *tool integration approach*. The tool should translate from a UML 2.0 specification to a quantitatively assessable simulation model in which the characteristics of the target environment are represented using facilities provided by the simulation library (or tool) used.

We also followed the XMI-based tool integration approach that is advocated in the UML Profile for Schedulability, Performance and Time[Gro02], involving two considerations. The first is to determine the *most suitable* library when translating from a UML 2.0 model to a simulation model. The second is to determine how a UML 2.0 model could be represented

using the chosen library and adapt the library to include features that are required for our primary objective. With these considerations in mind we review network simulation packages and select a suitable package for use in our performance analysis tool.

4.5.1 Requirements

Any simulation software falls into one of three categories [BCNN01]. These are general-purpose programming languages (e.g. C++, Java libraries), simulation programming languages (e.g. SIMULA, GPSS) and simulation environments. Simulation environments can be distinguished by cost, application area and animation capabilities.

In [BCNN01], Banks et. al. mention that very few people develop simulation software using general purpose programming languages alone. The benefit of developing models using programming languages are in gaining an understanding of how the basic simulation concepts and algorithms are implemented.

Here we discuss our requirements[5] in terms of features that candidate software may or may not have. For each simulation package that we examine, we discuss the support for each essential and desirable criteria listed below. Note that we only consider packages offering process-based discrete event simulation and some application area support.

It is also important to note that certain criteria are in conflict. For example, while it is desirable to use a simulation package that is as efficient as possible, we must also consider the fact that we will have to generate the simulation code and have limited resources. Generating code for a platform with a garbage collector (e.g. C#, Java) would require less development resources [Her03] than code in which the developer manages memory (e.g. C, C++). Hence the requirement of using efficient simulation software and using a language with a garbage collector are in conflict.

- **Process-oriented Discrete Event Simulation**: It is clear that in our application domain we are concerned with concurrent, communicating entities (processes) and as such we require *process-oriented* discrete event simulation software. When *process-oriented*, the event scheduler contains a central list of scheduled events with an associated pointer to a process [Hil01].

- **Programming Language**: Generating simulation code in which memory is managed by a garbage collector is *beneficial* with regard to development resources.

[5] Our primary sources are Jane Hillston [Hil01] (Chapter 11) and Banks et. al. [BCNN01]

- **Simulation software type**: since we will have to automatically transform a XML-based UML 2.0 model into a simulation model, a general purpose programming language or simulation programming language is required. In the case of open-source simulation software written in general purpose programming languages, an additional *benefit* would be to gain insight into the rudiments of discrete event simulation.

- **Cost and Technical Support**: free simulation software is available to academia and using such software is *preferred* since we have a limited budget. Free software may have a hidden cost however which comes in the form of a lack of technical support and bug-fixing.

- **Animation capabilities**: our model editor has superb animation capabilities and as such this is *not essential*. However, animation capabilities in the performance model would be *beneficial* for model verification and validation.

- **Extendability**: the ability to generate simulation code which accurately represents protocol software modelled using UML 2.0 is a *primary requirement*. By *extendable* we mean the ability to extend the features of the simulation software so that we are able to generate such code.

- **Application Area Support**: communicating systems specified using extended finite state machines (EFSM), as is done in SDL and Estelle, have functional signalling and timing abstractions. In addition, means of representing architectural aspects is provided in both languages. The ability to represent such architectural and behavioural abstractions with simulation package features would be *greatly beneficial* in constructing valid performance models.

- **Trace Routines**: trace routines are useful for both model verification and validation and the ability to add custom routines is *essential*.

- **Statistical Support**: good random number generators and a variety of random variable distributions are *required*. The distributions are particularly important in representing non-functional delay constraints, as is required for performance modelling.

- **Protocol and Application Model Libraries**: in modelling a communication protocol, models of expected service data unit traffic (application models) and lower level

protocols would be *extremely useful* in terms of model validation.

- **Wide User Base**: simulation software with a wide user base is *highly desirable* as it is more likely to be correct.
- **Output Analysis and Report Generation**: output analysis routines and graphical plots of performance metrics are *highly desirable*. In addition, automatic statistics gathering and reporting would be beneficial.
- **Efficiency**: with the inherent complexity of simulation programs and the length of simulation runs required for statistically significant runs, the efficient implementation of the simulation code is *of paramount importance*.
- **Complexity**: simulation languages may be difficult to learn how to use and hence the chosen language should *not be overly complex*.

4.5.2 Commercial Packages

CSIM

The CSIM simulation toolkit [Sof04][Sch01], by Mesquite Software, is a process-oriented, discrete event simulation package giving the modeller the ability to use standard C/C++ in model construction. CSIM is "an object module library together with C or C++ header files and example files" [Sof04]. The software is relatively inexpensive, a student license is available at $55 per seat.

CSIM is used as a teaching and research tool for studying network protocols. CSIM also has a library (OptQuest), that is specifically designed for performance and cost optimization. Since models are built in C/C++, CSIM should be readily extendeable and execution should be efficient. Mesquite Software advertise CSIM's fast execution and efficient models as one of its major features.

OPNET

OPNET Modeler, by OPNET Technologies, is said to be the leading network simulation modelling environment used in industrial R&D. Corporations using OPNET [Tec04] include Alcatel, Nokia, Ericsson, Samsung, Siemens, Sony, NASA and the US Department of Defence.

OPNET is a simulation modelling *environment* providing a variety of integrated components which include graphical editors, analysis tools and animation tools. The graphical editors are used to specify network, and network component, structure. The software is free to the academic research and teaching community.

With regard to extendability, individual network objects can be modelled at the process, node or network level. In addition any required behaviour can be simulated with C or C++ logic in finite state machine transitions. OPNET also supports *total openness* meaning that APIs for program-driven *construction* or *inspection* of all models and result files are provided.

OPNET provides extensive support for network protocol and other telecommunication development. Finite state machine modelling, hierarchical network models, various network link types (with associated delay and error characteristics), network device libraries and mobility modelling features are provided. OPNET also provides a comprehensive library of detailed protocol (e.g. TCP, IP, FDDI, Ethernet and 802.11) models that are provided as finite state machines with open source code. In addition, OPNET provides comprehensive support for protocol programming with hundreds of library functions for simplifying writing protocol models. Lastly, OPNET advertises itself as the most scalable and efficient simulation engine capable of simulating the behaviour of thousands of nodes.

4.5.3 Open-source Packages

Unisinos Simmcast

Simmcast [MB02], developed at Unisinos University, Brazil, is a simulation framework that is based on the JavaSim[6] simulation library. JavaSim was developed at the Department of Computing Science, University of Newcastle upon Tyne, and is a set of Java packages used for developing discrete event process-based simulation experiments.

Simmcast is open-source software and as such provides an opportunity to study the implementation of rudimentary discrete event simulation software concepts. Simmcast provides network software building blocks and primitives that are specifically designed to be easily extendable. "...the user needs to add or extend classes or interfaces of the framework according to the specific protocol and configuration being evaluated"[MB02].

Simmcast provides support for multicast protocol and unicast protocol simulation. An

[6]Since the features of JavaSim are provided by Simmcast we do not review JavaSim separately.

API providing a set of network *primitives* (typical timer and communication operations) is provided. With Simmcast the user specifies experiments using *building blocks* which are *node*, *thread*, *path*, *group*, *network*, *packet* and *stream*. Nodes are connected by paths which have associated properties such as bandwidth, packet loss probability and propagation delay. In addition, each node can have a sending and receiving processing delay which is used to model packet processing delay. State machine abstractions are however not provided.

Being written in Java, and with trace results being written to file, Simmcast is not intended for large simulation experiments. With regard to complexity, Simmcast is designed to be simple to use and extend.

OMNeT++

OMNet++ [Var04] is an open-source simulation environment with its primary application being the simulation of communication networks and protocols. It is actively used in both the scientific and industrial communities. OMNET++ is open-source software that can be modified and is freely available for noncommercial users. It is actively used in research environments and is supported by an online community [Var04]. With OMNET++ components are programmed in C++ and then assembled into larger components using a high-level language[7], called NED. The simulation kernel can be embedded into third-party applications and extensive GUI support (NED editor, graphical output vector plotting tool, simulation execution GUI) is provided.

With regard to extendability, either a threading programming model or finite state machine (FSM) programming model can be used with OMNeT++. Any level of model detail can be expressed by the user with either programming model. Network topology is either defined graphically or using an equivalent textual format. In such a hierarchically nested network topology, *nodes* are connected using *links*, or connectors, and sets of nodes form *modules*. As with Simmcast, optional parameters associated with links include propagation delay, bit error rate and data rate. OMNeT++ provides support for the specification of FSMs by providing a class, a set of macros and an API to build FSMs. Importantly, these FSM's are said to operate "very much like OPNET's or SDL's"[Tec04].

OMNeT++ offers a few protocol models, however these cannot [Var04] compete with the large selection offered by commercial tools such as OPNET. Currently twenty-seven OMNeT++ communication network models are provided by the community and include

[7]Simmcast takes an almost identical approach.

a mobility framework, an IPv6 protocol Suite, Ethernet models, Peer-to-Peer swarming protocol models, and the Real-time Transport Protocol (RTP).

Berkeley NS

The Network Simulator (NS) [oSC04b], is aimed at network research with an emphasis on transport layer, routing and multicast protocols. A significant amount of research [oSC04a] has been conducted on NS itself and in addition NS is often used in academia when experimenting with enhancements to TCP. NS is implemented in C++ and runs on several Unix variants (FreeBSD, Linux etc.) as well as on Windows. It relies on various packages which include Tcl/Tk, OTcl (Objec-oriented Tcl) and Nam-1 (a Tcl/Tk network simulation animator).

With NS protocol simulation experiments are set up using OTcl and C++. Experiment parameters and configurations are changed using OTcl interface and protocols are written in C++. Settings that are specified using OTcl include the type of event scheduler, packet formats, node creation and network topology. Although NS is complex (and hence may have a steep learning curve) the fact that it is open-source and has been heavily used in research (with supporting documentation) may offer compensation.

With NS, *agents* are used as endpoints for the construction and consumption of network packets (in particular IP packets) at *various layers*. NS provides a comprehensive *Agent API* with network primitives which mimic *socket APIs*. It seems plausible that the APIs provided by NS can be extended to represent UML 2.0 protocol models. With NS architecture is specified using *Nodes* and *Links*. Nodes, and the components of nodes, are specified using OTcl. Each node has an address, a list of neighbours, a list of agents, a type identifier and a routing module. NS does not have explicit finite state machine support as is the case with OMNeT++ and OPNET.

NS supports a variety of protocol and propagation models: "Almost all variants of TCP, several forms of multicast, wired networking, several ad hoc routing protocols and propagation models (but not cellular phones), data diffusion, satellite" [oSC04b]. Most of the protocol models supplied with NS have been validated.

4.5.4 Network Simulation Package Selection

In this section we have examined both commercial and open-source discrete event simulation software. Each simulation package was evaluated by investigating the extent to which a set

of requirements were met, with a focus on the *software type*, *extendability* and *application area support*. Admittedly, our approach to the evaluation of the simulation packages is somewhat subjective and would ideally have been based on experience using the individual packages in our context.

An important consideration in our evaluation of each package was that one of our aims (see Section 1.3) in this work was to translate from a UML 2.0 protocol specification to a simulation model. In other words, we have to automatically *generate* the code representing the protocol specification using the output (XMI) of a UML 2.0 editor. Lastly, a consideration was that using an open-source solution has the benefit of offering the opportunity to learn about the implementation of rudimentary process-based discrete event simulation software.

With the above mentioned requirements and considerations in mind we examined CSIM19, OPNET, Simmcast, OMNeT++ and NS. When comparing CSIM19 and OPNET (both of which are commercial solutions), it is clear that OPNET meets our requirements to a greater extent in particular due to its support for finite state machine (FSM) models, a comprehensive library of protocol models and its wide user base. The support for the specification of protocol logic using a finite state programming model is significant, since with the version of UML 2.0 we are using, behaviour is specified using SDL state machines.

In our investigation of open-source simulation packages both OMNeT++ and NS were shown to be suitable for our purposes, although NS has the benefit of having a larger academic user base and thus a far greater amount of protocol model libraries. OMNeT++ had the benefit of providing specific support for the specification of FSM models and an associated FSM API.

In conclusion, *we decided to use Simmcast* as the most suitable simulation library since it met what we considered to be our most important requirements to the greatest degree. Simmcast had the benefits of relative simplicity, a moderate[8] degree of application domain support, being open-source and importantly it was build using Java. In addition, with Simmcast the process-interaction is not hidden from the modeller's view. Even though *execution* efficiency is important in any simulation software, in this project we had *development* efficiency requirements and considered the code generation complexity involved when using Java to be significantly less than when using C or C++ in which memory allocation problems (memory leaks, crashes, heap corruption) are of concern. In a similar project

[8]FSM modelling support is not supported in Simmcast.

with greater development resources OMNeT++, NS and OPNET may be more attractive alternatives.

4.6 Summary

In this Chapter we have investigated a number of aspects regarding the use of simulation to predict communication system performance. We firstly motivated the use of simulation modelling (rather than analytic modelling) in our context and briefly investigated the use of process-based discrete event simulation to model protocol execution. The use of simulation was found to be preferable primarily when

- analytic abstractions and assumptions are not appropriate
- the transient behaviour of a system is important
- the size of the state space generated using analytic modelling is too large to be stored.

In our brief investigation into the use of process-based discrete event simulation, we saw that simulation packages must be used with care since the process-interaction is hidden from the modeller's view. Therefore modellers using simulation packages must have both a basic understanding of the concepts (of process-based discrete event simulation) and a detailed understanding of the hidden rules of operation of the simulation package.

We then investigated model verification and validation. Although validation is imperitive when in performance modelling, in this work we take the *tool builder* perspective rather than the *tool user* perspective. In other words, our primary concern is with model verification and in the application of the techniques listed in Section 4.4. Finally, we reviewed commercial and open-source network simulation packages and selected Simmcast as our simulation package.

Chapter 5

A Methodology for Protocol Performance Engineering with UML 2.0

5.1 Introduction

In this chapter we develop and discuss the proSPEX methodology, semantic time model and high-level architecture. With proSPEX our overall goal was for the automated performance analysis of communication systems using a tool integration approach. As we have seen in Chapter 2, the automated functional and performance analysis of communication systems specified with some Formal Description Technique has long been the goal of telecommunication engineers. In the past SDL, Estelle and PROMELA have been the most popular FDTs for the purpose. With the growth in popularity of UML the most obvious question to ask is whether one can translate one or more UML diagrams to a performance model. In Chapter 2 and 3 we saw that in the past UML has been unsuitable due to restrictions such as semantic variation points and a lack of a syntax. However, with UML 2.0 released and using ITU Recommendation Z.109, the abstractions, tightened semantics and syntax that are found in SDL become available.

Our first consideration in this chapter is the development of a methodology which entails both identifying the roles of UML 2.0 diagrams[1] in the performance modelling process

[1]Detailed coverage of UML 2.0 is beyond our scope, although we do provide a brief introduction to the language in Appendix A.

and refining our requirements as stipulated in Section 1.3. In addition, questions regarding the means of specifying non-functional duration constraints[2], which are essential in performance modelling, must be considered. In Section 5.2 we formulate a methodology for the design and performance analysis of communication protocols specified using UML 2.0. This methodology has two primary sources of influence. The first source of influence is related work, which we investigated in Section 2.6.2, in which approaches to universal SDL performance modelling issues are addressed. The second source of influence is the UML Profile for Schedulability, Performance and Time [Gro02] which advocates an XMI-based tool integration approach. In our methodology we therefore discuss the construction of a model processing tool that uses the features of an extendable simulation library, namely Simmcast.

We detail the semantic time model associated with proSPEX in Section 5.3 and then discuss the high-level architecture of proSPEX in Section 5.4. In our discussion of the proSPEX tool architecture we provide an overview of Simmcast as well as the extensions that are needed to Simmcast in order to make it suitable for our purpose.

5.2 The proSPEX Methodology

The proSPEX methodology is for the modelling, verification and performance evaluation of communication software and is presented in Fig. 2. In developing a methodology our goal was to specify protocol architecture, behaviour and environmental characteristics using a minimal subset of diagrams. Hence a particular subset of diagrams are translated to the executable performance model while other diagrams are used purely for understanding and communication between developers. The purpose of our methodology is to serve as a guide of how to go about protocol design, specification and importantly the specification of non-functional delay constraints and subsequent performance analysis. In addition, our methodology serves as a *refinement of our proSPEX requirements* as defined in Section 1.3.

In the design stage we advocate the use of accepted protocol engineering best practises [LTB98][PT00] and in the performance engineering stage (the specification of environmental constraints) we consider both the features of our chosen simulation framework and previous work where SDL is used as the specification language.

[2]Non-functional duration constraints are used for specifying temporal aspects which are relevant in performance modelling.

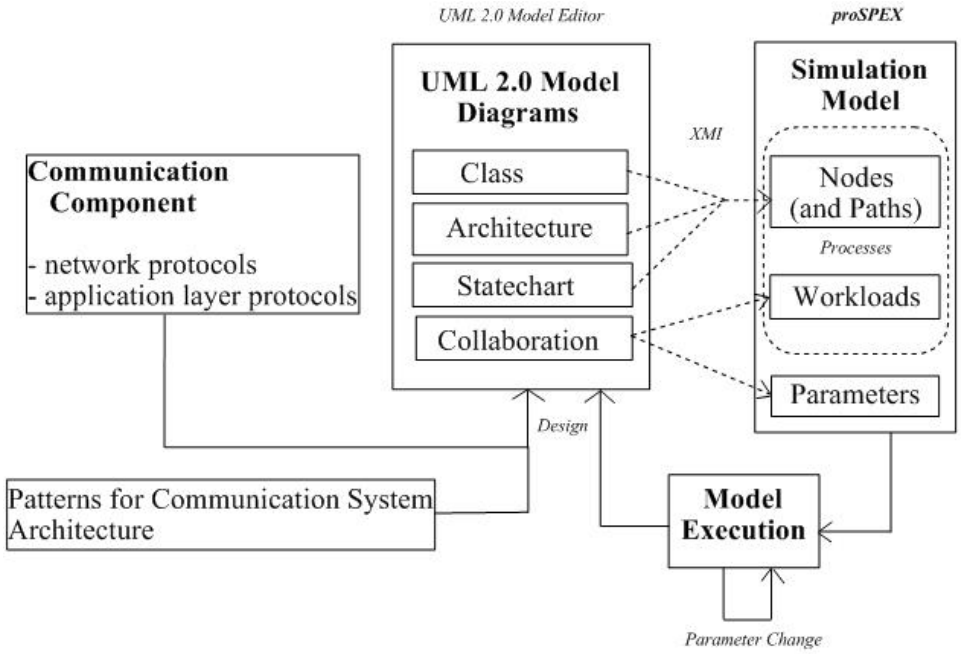

Figure 2: The proposed methodology supported by the simulation-based proSPEX performance analysis tool

Requirements Definition

The first step is to establish the requirements of the communication component. In the case of a transport layer protocol a requirement may be to use the available bandwidth as efficiently as possible. Following requirements definition we identify[3] or design suitable network and application layer inter-component protocols. UML 2.0 use case and sequence diagrams could be used to aid understanding but these are not used when generating the simulation model, as can be seen in Figure 2.

Architecture Specification

The next step is to use a combination of UML 2.0 class and architecture diagrams to design the protocol architecture. The use of design patterns for protocol system architecture[4] [PT00] is recommended at this stage. The focus of this stage is to identify the active classes and their interfaces.

[3]Requests for Comments (RFC) documents could be used here.

[4]We investigate patterns for protocol system, entity and behaviour specification in Appendix B.

Interface-based design has the benefit of both reduced design complexity and giving distributed teams the ability to work concurrently while using the interface as a contract. In UML 2.0 an interface is a classifier representing a declaration of a set of public features and obligations[Gro03]. Interfaces are not instantiable, instead they are either *provided* or *required* by a classifier such as a class. When a class provides an interface it carries out its obligations to clients of instances of the class. When a class requires an interface it means that it needs the services specified in the interface in order to perform its function and fulfill its obligations to its clients. The notation introduced for a provided interface is a full-circle lollipop whilst the notation introduced for a required interface is a semi-circle lollipop.

Figure 3 shows the architecture diagram of an active class with two *parts*, namely any number of Sessions and a single RoutingPeerProxy. The parts are linked with *connectors* that are attached to *ports*. Note that notationally ports are the squares to which the required interfaces, provided interfaces and connectors are attached. Each *port* serves the duel purpose of being used to group an active class's related interfaces and also acting as interaction (or connecting) points through which the services of a class can be accessed. In the architectural view of an active class we want to be able to distinguish between behaviour that is delegated to the class itself and behaviour that is delegated to its parts. Connectors terminating in a behaviour port mean that the signals sent to the port are handled by the containing class. Notationally a behaviour port is represented by a state symbol attached to a square port symbol, as can be seen in Fig. 3.

Behaviour Specification

Following the architectural specification we specify the detailed behavior of active classes by implementing state machines using statechart diagrams. We use specialized communication abstractions derived from SDL in this model-driven development process. Fig. 4 shows a part of a UML 2.0 statechart diagram. Note that the syntax used is the Telelogic Tau *UML Syntax* derived from SDL. Once this stage is complete the software is verified using facilities provided by the model editing tool, in our case Telelogic Tau G2.

Behavioural verification using Tau is done by firstly generating a C code executable program from the UML model. The generated program is linked with a run-time simulation library which provides automatic and manual (tracing execution by inspection) simulation modes. The execution is controlled either via a user interface or by using a set of console commands. The result of model execution can be logged as either a textual trace or sequence

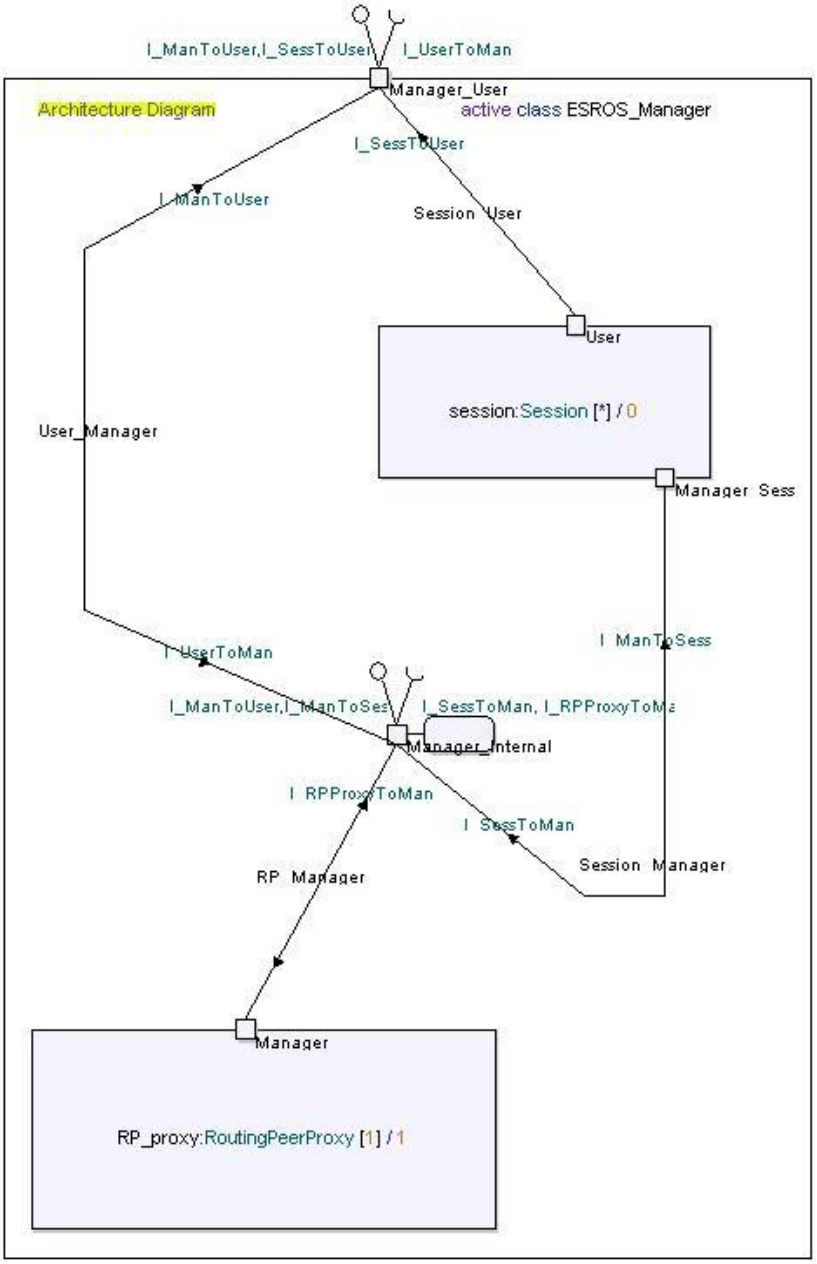

Figure 3: Architecture specification with UML 2.0

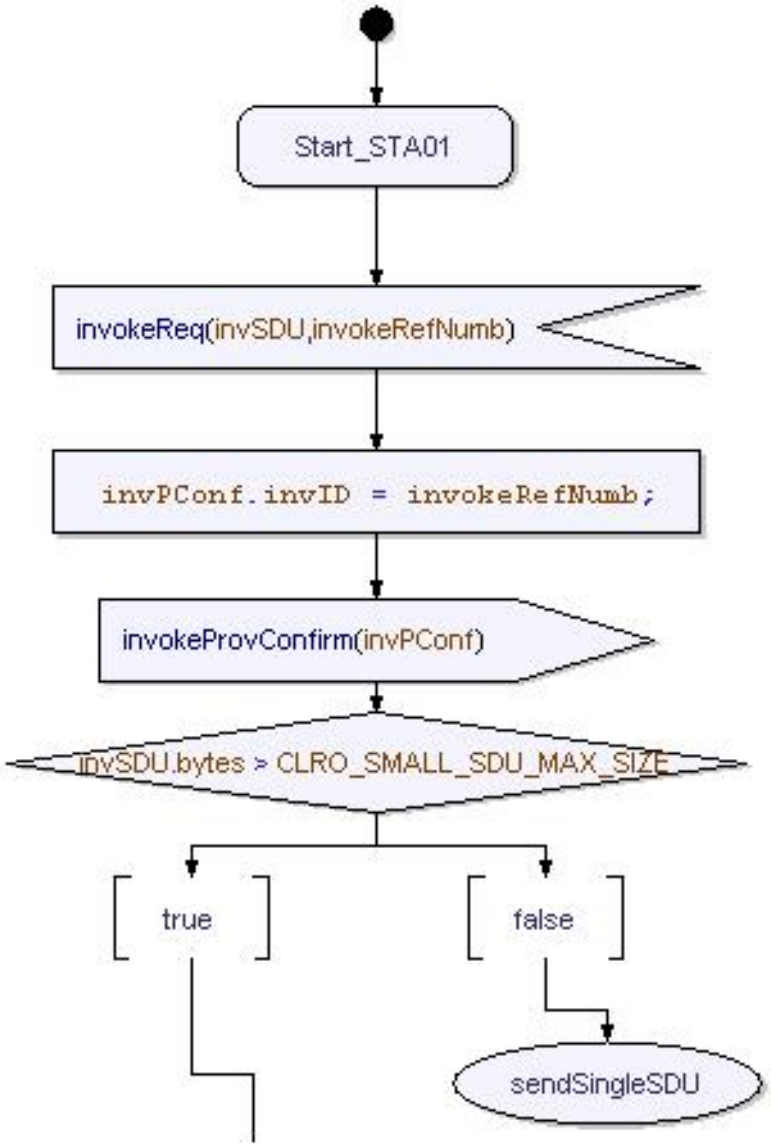

Figure 4: Behaviour specification with UML 2.0

diagram. Users are able to monitor variable values and execution steps using simulation sessions.

Telelogic has named their execution tracer (or animation facility) a *Model Verifier*. This is not entirely correct since they are, in essence, providing sequence diagram simulation tracing. In other words exhaustive verification by state space exploration is not provided and neither are model-checking facilities as one would find when using PROMELA and SPIN.

Simulation Scenario Specification

Once the software has been debugged using the Tau model execution facilities the performance modelling phase commences. With proSPEX non-functional timing annotations are embedded in UML 2.0 comment symbols, as is done in the case of SDL performance analysis tools such as objectGEODE [Rou01]. This allows for the use of commercial modelling tools and, as we discussed in Section 2.6.2, using such an approach is important when one wants a tool to be useful to the largest possible audience.

The performance modelling phase starts with the modelling of the environment of the communication component. That is, we create client and server, or peer, active classes and their associated state machines. A collaboration diagram (see Figure 5) is then drawn up illustrating a simulation scenario which, in combination with the statechart diagrams of the client(s) and server(s) serve, as the workload.

This collaboration diagram would indicate the number of clients and servers and also network link characteristics (loss probability, bandwidth and delay distribution). Processing delay timing constraints, or delays, are associated with active classes and may be deterministic or randomly distributed. The network link and processing delay parameters are specified using comment symbols.

Once the scenario has been completed the proSPEX tool user imports the model from which a semantically equivalent simulation model is generated.

Results

The events and corresponding trace messages that the simulator is able to generate dictate the set of performance statistics that can be calculated. In determining a suitable set of traceable events we use those that are found in the SPECS tool [BMSK96][BMSK95]

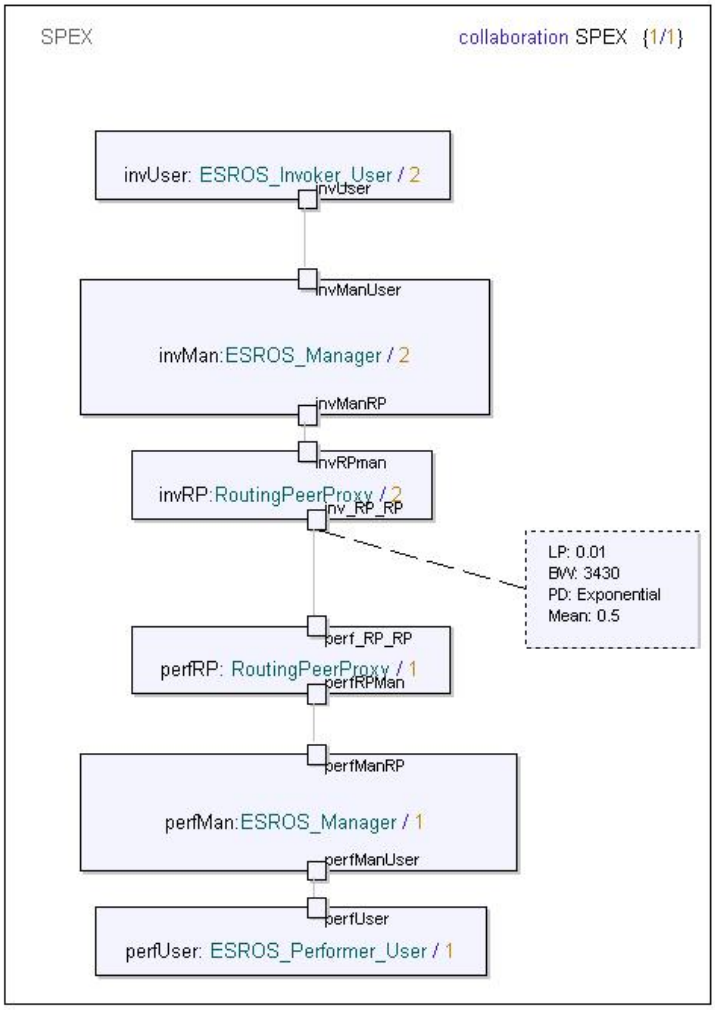

Figure 5: Simulation scenario and workload specification

[dVHVZ96]. Hence the simulation model generated by proSPEX should be able to generate the following types of trace messages[5] for some general time t_i:

1. Message M sent via Connector C from process P1 to process P2 at time t_i
2. Message M from process P1 read by process P2 at time t_i
3. Message M from process P1 arrives in queue of process P2 at time t_i
4. Process P created at time t_i
5. Process P destroyed at time t_i
6. Overflow: message M from P1 to P2 discarded at time t_i
7. Process P has transition from state S1 to state S2 at time t_i
8. Message M from process P1 discarded by process P at time t_i
9. Timer T set to duration d in process P at time t_i
10. Timer T reset in process P at time t_i
11. Timeout: Timer T in process P at time t_i

The performance measures that can be calculated from the analysis of simulation traces containing the above mentioned messages include the following:

1. **Mean queue waiting time**: this is the average time that a signal spends in the queue of a process. A high mean queue waiting time means that *process response time may be too slow* or that there are too many retransmission messages in the queue of a process as a result of the *timeout of the sending process being too short*. Trace messages 2, 3 and 8 are used in the calculation of this statistic.
2. **Connector throughput**: this is *the traffic on a connector* and trace message 1 is used in its calculation.
3. **Mean and maximum queue length**: The buffers of a communication system are often modelled using process queues. A high maximum queue length indicates that the *system requires large buffers*. Trace messages 2, 3 and 8 are used to calculate this statistic.

[5]For brevity we use the term *process* instead of *active class*

4. **Detection of queue overflows**: queue overflow is indicated by trace message 6 and shows that the *process' buffers are too small*.

5. **Throughput of a state**: this statistic shows how many times a state is reached and hence which program parts are frequently processed. States with a *high throughput may indicate process bottlenecks*. Trace message 7 is used in the calculation of this statistic.

6. **Discarded signals**: Such signals may either be caused by *insufficient process buffer size or as a result of being sent to processes that no longer exist*. Trace messages 6 and 8 show discarded signals.

7. **List Unreachable states**: Process' states that are never reached indicates dead code however since simulation is not exhaustive it is not guaranteed that the code is actually unreachable. Trace message 7 is used in determining unreachable states.

8. **Average time spent blocked in a state for a signal**: This time period shows the *idle time of a process* and records how long a process spends waiting for a signal. Trace message 7 is used in its calculation.

9. **The lifetime of a process**: This statistic is used in other statistical calculations and uses trace messages 4 and 5.

10. **Timeout reset and expiration ratios**: The first ratio is that of the number of timeouts set to the number of timers expired. It shows what proportion of timeouts were exceeded. The second ratio is that of the number of timeouts in a queue to the number of timers set. It shows how many timeout messages in the queue had to be reset. Both ratios *show whether timeouts in the system are set at sub-optimal values* and are very useful in improving the performance of protocol systems. Trace messages 9, 10 and 11 are used in the calculation of the mentioned ratios.

Naturally any analysis results would refer to either the steady-state or transient behaviour of the system and would be computed with confidence intervals. These measures would then prompt the user to either change the simulation parameters or the model itself.

5.3 The proSPEX Semantic Time Model

When building performance models using UML 2.0 enhanced with the Z.109 profile, previous approaches to performance analysis that incorporate temporal aspects into SDL specifications (examined in Section 2.6.2), can be applied. Each approach has an associated semantic time model and means of specifying non-functional duration constraints. Here we explicitly mention how these non-functional time related aspects are represented in proSPEX. The proSPEX semantic time model is naturally influenced by the use of a tool integration approach[6] in which the facilities of a simulation framework are used where deemed appropriate. This approach is similar to the SDL/OPNET[7] approach [MHSZ96] which entails a mapping from SDL descriptions (annotated with constructs for describing delays, processing resources and workloads) to executable OPNET models.

- **Communication delay**: with proSPEX communication delay is associated with packets that traverse network links. Such links have an associated propagation delay (modelled with a random distribution), bandwidth and loss probability. When a packet (or signal) is sent across a network link, delay is applied in two stages before it reaches the receiving queue of the target process. In the first stage the packet is delayed by a *sending time*, which is calculated using the bandwidth and a packet byte size attribute. In the second stage a *propagation delay* is applied and loss probability is applied, with the propagation delay being drawn from a random distribution.

  Erlang, exponential, hyperexponential, normal and uniform delay distributions can be used to model propagation delay. These delay distributions are provided by Newcastle SimJava, which Simmcast is based upon. In Section 2.6.2 we saw that the definition of time in the dynamic semantics of SDL is loose in the sense that it acknowledges that the system will execute in real time with delays on channels, but does not specify how the system execution is affected by this constraint [BMSK95]. By explicitly modelling propagation delay using delay distributions we tighten the dynamic semantics of SDL.

- **Processing times**: with proSPEX we use the facilities of the underlying simulation framework to associate a *sending* and *receiving* processing delay with active classes.

[6]The reader is strongly encouraged to re-visit the aims of this project, stated in Section 1.3, at this stage. Our goal was *not* to include all possible features in our tool and associated semantic time model or to develop a novel semantic time model. We have used the features of the underlying simulation framework in the proSPEX prototype.

[7]OPNET is a network simulation environment that is discussed in Chapter 4.5.

Active classes that have such delays specified have their execution blocked by the specified delay values whenever a signal is sent and received. In other words the sending and receiving delay associated with an active class is effectively mapped to each input and output operation. Such delay is deterministic by default, although the use of random delay distributions is possible. The choice of distributions is the same as those that are used for the specification of propagation delay.

In our examination of previous approaches to performance analysis using SDL, we saw that the problem of performance analysis using SDL had already been extensively researched, as is mentioned by Mitschele-Thiel and Muller-ClosterMann in [MTMC99]. We saw that processing delay can be associated with actions, transitions or entire processes. In our case, one of our objectives was to determine to what extent the current version of UML can be used for performance analysis and also to use a tool integration approach. Thus, in modelling processing delay, we have used the features of the underlying simulation framework in our initial prototype.

- **Execution modes**: with execution modes we consider time passage in parts of the system with no time delays expressed. With proSPEX only input and output actions are time constrained, and all non-time constrained actions are immediate.

- **Time delays on the external environment**: with proSPEX the environment is modelled by processes in which signal characteristics can be expressed using time guards.

- **Scheduling**: scheduling information is not represented with proSPEX and is controlled by the underlying simulation scheduler. Process scheduling is determined by the order of the individual process event sequences. The representation of scheduling information could be considered in future work.

5.4 The proSPEX Tool Architecture

A general overview of the proSPEX tool architecture is shown in Figure 6. With proSPEX our intention was to create a model-processing tool and not a model editor since developing an editor would deviate from the primary objective of the project. Telelogic Tau G2 offered an XML-based model file format which was sufficient for our purposes, although the standard XML Metadata Interchange (XMI) 2.0 file format would have been preferable,

since this would in principle allow any future UML 2.0 editor to be used. As we can see in Figure 6 the code generation process involves filtering the Telelogic Tau XML and placing the filtered aspects into data structures that can be used for simulation code generation.

We investigate the features of Simmcast in Section 5.4.1 and then identify necessary extensions and changes to Simmcast in Section 5.4.2. These changes are needed in order to realize the automated mapping from a UML 2.0 protocol model, specified in accordance with the proSPEX methodology, to an executable representation.

5.4.1 An Overview of Simmcast

We aimed to generate discrete-event simulation programs by using the features of an existing simulation library. A review of the available simulation packages, conducted in Chapter 4.5, showed that Simmcast[MB02], an object-oriented framework for network simulation, would be ideal. Simmcast is specifically intended to be used in research environments with limited resources, as the excerpt from [MB02] shows:

> ...the complete development of a dedicated simulation tool from scratch is not practical, since the amount of resources dispensed in such a project would detract the researcher's focus from the project.

Simmcast offers extensible building blocks that are combined to describe the simulated network environment. The fundamental building blocks are *node*, *thread*, *path*, *group*, *network*, *packet* and *stream*. Nodes, each of which are uniquely identified by an integer and containing at least one thread of execution, are the fundamental interacting entities and are connected via paths. The user extends the Node class, via inheritance and places protocol logic and a set of primitives in the extended class. These primitives are a set of typical communication and timer operations and are defined as an API. They include `send` (used to send packets), `receive` (a blocking packet receive operation), `tryReceive` (a non-blocking receive operation), `setTimer` and other operations.

As mentioned above nodes are connected by paths which may represent a packet queue, a physical link or a logical path between two nodes. Hence the meaning of paths and nodes is defined by the modeler. A path between two nodes is represented by a path queue which holds messages in transit. Each path is parameterized with a bandwidth value, a packet loss probability value and a propagation delay distribution. In the case of paths connecting

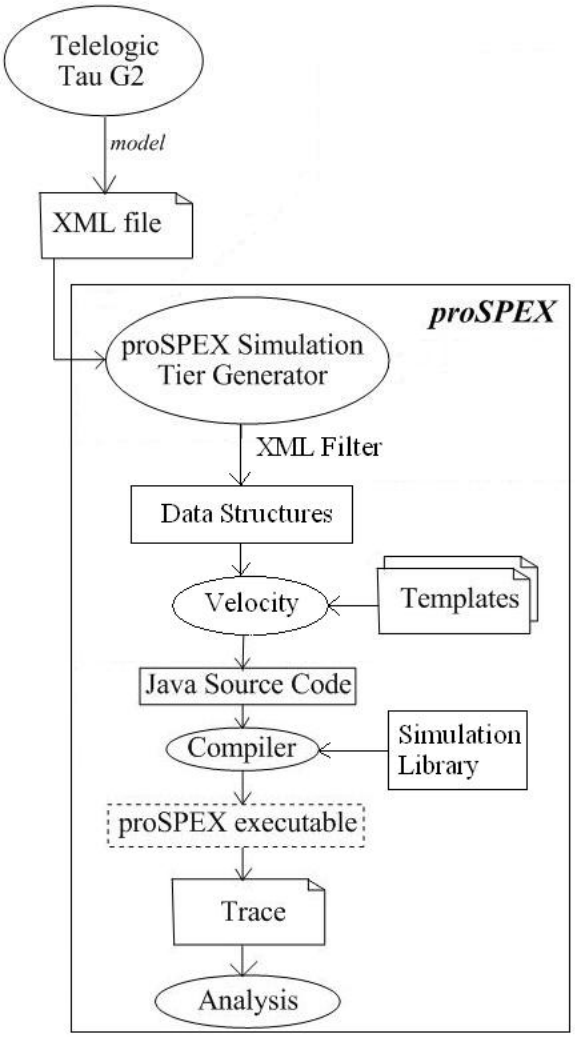

Figure 6: The proSPEX architecture

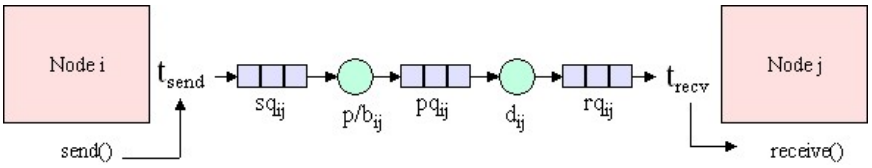

Figure 7: Conceptual simmcast packet flow model with service times.

nodes in adjacent protocol layers, in the same machine, the bandwidth is set to infinite, the delay zero and loss probability zero.

Figure 7 shows a conceptual model of packet flow in Simmcast. When the `send()` primitive is used to send a packet from node i to j, the first step is for a t_{send} delay to be applied. This delay parameter, which is deterministic in Simmcast, is associated with node i and models processing delay prior to a packet being sent. Following the t_{send} delay, the packet enters the sending queue of node i, sq_{ij}. The sending queue can be set to the default unlimited capacity or a specific size. Once the packet in question is at the head of the sq_{ij} queue the packet size, p and bandwidth, b_{ij}, are used to calculate the *sending time delay*, or time taken to be admitted to the propagation queue. Note that the loss probability is applied directly prior to the packet being admitted to the propagation queue.

The propagation queue, p_{ij} is virtual in that it is not directly instantiated in an actual implementation. A propagation delay, d_{ij}, which is a value taken from a random distribution (e.g. exponential) is applied to the packet once the simulation scheduler has scheduled it to be propagated. Following the d_{ij} delay the packet is inserted in the node j's receiving queue, rq_{ij}. Lastly, a receiving processing delay t_{recv} is applied prior to the packet being processed when the protocol logic in node j's thread executes a `receive()` call. The simulator itself does all of the above transparently, the modeler simply calls `send()` and `receive()` in the respective nodes.

In order to allow for the calculation of performance metrics the Simmcast architecture generalizes different forms of accounting into groups of event categories, which are *traceable events*. Such events are considered to be when a packet is enqueued and dequeued from the queues depicted in Figure 7 (sq, pq, rq and tq). Tq is a simulator queue used for the scheduling of future asynchronous events which are typically timer expiration (or cancellation) and associated event handling. Other types of trace events can be easily added to Simmcast since it provides a unified output interface (via the `TraceGenerator` class) where all the simulation events are reported.

With Simmcast the first stage in the creation of an experiment is to construct the simulated protocol by extending the *node* and *nodethread* classes. After having constructed the simulated protocol, the second stage is to specify a series of model parameters and the network topology using a separate simulation description file. The simulation description file could be seen as the equivalent of the main file of an executable, except there is no Java code in the file, a custom simplified syntax is used in the description file. An example of a Simmcast simulation description file is given in [MB02].

5.4.2 Extensions Needed to Simmcast

From our overview of Simmcast in the previous section it is clear that extensions to Simmcast are needed in order to be able to generate the simulation Java code that represents a UML 2.0 protocol model. Such a model would have been specified in Tau G2.1 in accordance with the proSPEX methodology. Such extensions to Simmcast would be designed to represent the UML 2.0 model using the facilities of the resultant framework. Once Simmcast has been extended in a *proSPEX* library, we can use a text-templating engine to generate simulation code, having filtered the Telelogic Tau XML. The extensions to Simmcast, that *we have identified as being required*, are listed and discussed below. The actual implementation of these changes is discussed in Chapter 6, "The proSPEX Implementation".

Finite State Machine Representation

As mentioned in Section 5.2, in the proSPEX methodology finite state machines (FSMs) are used to represent protocol behaviour. Since Simmcast does not have built-in state machine abstractions we must extend Simmcast for the purpose. The subject of finite state machine representation in Java (and other object-oriented languages) has been thoroughly investigated [vBB99][HBR00].

In [vBB99], we see that the *states*, *events*, *transitions* and *actions* are the core parts of an FSM that must be represented in an implementation. Approaches to implementing FSMs in OO languages include the application of the State Pattern [vBB99], the application of the *van Gurp approach* [vBB99] or the by using case statements. The use of the State Pattern is said to be preferable since procedural FSM implementations in which case statements are used suffer from maintenance problems. Maintenance, or change, to an FSM implementation, such as when adding a state or transition, can be problematic when it is difficult to incorporate the change. The difficulty in making changes is influenced not only

by the number of classes that have to be modified but also by the degree to which FSM concepts are explicitly represented in the implementation.

From [vBB99] it is clear that the implementation of state machines is non-trivial and requires some thought. The main issues [vBB99] to be considered are:

1. *FSM evolution*: changing the structure of state machines over time, which is often necessary, is difficult using most implementation approaches.

2. *FSM instantiation*: since FSMs may be used several times in a system, techniques that prevent object duplication can be applied in order to conserve resources.

3. *FSM data management*: actions in transitions change data which has to accessible to *all* transitions in the FSM. This means that all variables have to be global which leads to maintenance issues.

In our case an additional issue, which is not listed above, is *FSM code generation*. That is, we are interested in an FSM implementation approach which would ease the complexity of the code generation process. It is not clear whether using an FSM implementation approach which would ease FSM evolution (making changes to an FSM) would in turn lead to less complexity in the proSPEX code generator. Despite this lack of clarity, we consider the approaches to resolving the FSM evolution, instantiation and data management issues investigated in [vBB99].

Removal of the Simulation Description File

As was mentioned earlier in Section 5.4.1, in Simmcast a simulation description file is used for network protocol simulation *parameterisation* and *architectural specification*. The simulation description file is "simply a text file with a series of constructor and method calls to be performed by Simmcast" [MB02].

The syntax used in the simulation description file was created by the Simmcast author and is relatively simple. The parameters that are specified include the type of simulation trace generator used, the number of client and server nodes, the inter-node link characteristics (e.g. delay distribution, bandwidth) and the specification of groups (useful in the case of multicast protocols).

The reasons for the use of a simulation description file, as opposed to a Java *main* file, is that it is seen [MB02] as being both a means of maintaining the simplicity of the system

and also of maintaining a "separation of concerns". Such a separation of concerns, in which topology and startup parameters are in the description file, and protocol logic is maintained separately in Java source files, is also used in NS-2 [oSC04b] and OMNeT++ [Var04], as we have seen in Chapter 4.

In our case however, UML 2.0 (and SDL) diagrams are used for the specification of topology and startup parameters. The specification of architecture and behaviour, using FDTs with graphical representations specifically designed with communication abstractions, is clearly superior to a textual description. We would clearly not gain any additional benefit from the "separation of concerns" derived from a simulation description file.

In addition, a major drawback of the Simmcast simulation description file is that no support for dynamic node creation is supported. That is, the entire topology must be specified in the simulation description file. We would thus have to build means of supporting dynamic node creation (and hence topology changes) into Simmcast.

We would therefore use an approach in which the simulation description file is not generated from the simulation description file. Instead, we will change Simmcast so that all parameters and topology information could be specified in the simulation main file.

SDL Pid Expression Representation and Implicit Addressing Representation

With SDL, each process (active class) has access to four Pid (process identifier) expressions which are means of accessing implicit process attributes. These expressions are *self*, *sender*, *parent* and *offspring*. Their meaning is intuitive, for example the *sender* expression returns a Pid value which refers to the process this instance received its most recent signal from.

We must allow for the use of these common expressions and build means of keeping track of these variables in the state machines that are generated by the proSPEX generator. In addition, the Pid expressions would be translated to the equivalent network addresses used in Simmcast.

Another aspect that we must address in proSPEX is implicit signal addressing. With SDL, signal addressing is mostly implicit meaning that it is not necessary to specify the destination address of a signal when using the *send* action. Explicit signal addressing is not necessary since the destination of the signal is deterministic from the system architecture. In an implementation, and with Simmcast, each signal that is sent must have a network address associated with it. Thus, we must determine means of translating from the higher level SDL signalling abstractions to a lower level Simmcast abstractions.

Architecture Representation and Specification

With Simmcast paths between nodes have to be explicitly set up in the simulation description file. As we explained earlier, we have identified the need to place topology and startup parameters in the main Java source file and not in a description file. Apart from extending Simmcast for this purpose, we must encode the protocol system architecture in our proSPEX extension to Simmcast. The reason for this is because we allow for dynamic node creation and termination. For example, a Manager process, which spawns *Session* child processes may do the following upon receiving a signal from a service access point:

```
new Session;
output invokeReq(invSDU,invokeRefNumb) to offspring;
```

In the example given above, a Manager process creates a Session child process and then sends an `invokeReq` signal to the child using the offspring Pid expression. In our proSPEX extension to Simmcast, some work would have to be done when the `new Session` statement is encountered. This would involve consulting the system architecture (which had been encoded in some data structure) and determining the nodes to and from which paths would have to be created. One aspect that we have to consider is that with Simmcast paths are unidirectional. So if two nodes send messages to each other two paths are created, one for each direction of communication. Thus, when encoding the architecture, we need to record which nodes may send messages to the created node and which nodes may receive messages from the created node.

Additional Trace Events

As mentioned in Section 5.2, we aimed to extend Simmcast to allow for the generation of a particular set of trace events. Due to the modular nature of Simmcast this extension should be fairly straightforward.

5.5 Summary

In this chapter we have reported on the various parts and facets of the proSPEX tool. Our first step was the development of a methodology in which we identified the roles of UML 2.0 diagrams in the specification and performance modelling process. In our methodology

we incorporated aspects and recommendations from both previous work in which SDL is used as the specification language, which we discussed in Section 2.6.2, and the UML Profile for Schedulability, Performance and Time (UML-RT), which we discussed in Section 3.5. In our methodology we used class, architecture and state machine diagrams for the specification of protocols while using collaboration diagrams for the specification of annotated non-functional duration constraints.

With the proSPEX semantic time model we have used the facilities of our simulation framework to represent non-functional time related aspects. In this model communication delay is associated with packets that traverse network links. Randomly distributed delay, transmission delay and a loss probability are applied to each networked packet. In addition, each node in the architecture has an associated sending and receiving processing delay. Performance annotations are all specified using an annotated approach and simple syntax.

In Section 5.4 we developed the architecture of the proSPEX tool which was to realize our methodology and semantic time model. We identified a set of extensions to Simmcast that are needed in order to be able to generate the simulation Java code that represents a UML 2.0 protocol model. These extensions included those for finite state machine representation, the removal of the simulation description file, SDL Pid expression representation and implicit addressing representation, architecture representation and specification, and lastly, adding necessary trace events.

Chapter 6

The proSPEX Implementation

6.1 Introduction

In this chapter we show how we addressed the proSPEX implementation issues and challenges, that arise in mapping from Telelogic UML 2.0 to a simulation model. Such challenges include ways of representing signal addressing, the encoding of the protocol system architecture and state machines. In addition, the proSPEX Tau filter and code generator implementations are discussed.

With the proSPEX implementation we attempt to realize the proSPEX methodology and semantic time model mentioned in our aims and investigated further earlier in Chapter 5. The proSPEX implementation has three distinct parts which are illustrated in Figure 8. The first part is the *proSPEX extension to Simmcast* in which means of representing finite state machines, architecture, additional trace events, SDL addressing and other aspects are implemented. The second part is the *proSPEX Tau filter, code generator and associated templates*. While the third part is a rudimentary *proSPEX GUI*.

6.2 Mapping from Telelogic UML 2.0 to a Simulation Model with the proSPEX Extension to Simmcast

In this section we discuss the implementation of those Simmcast extensions that we have identified as being necessary in order to map from a Telelogic UML 2.0 model to a simulation model. Our approach was to extend Simmcast and then to hand-code the ESRO Invoke Service protocol (which we had specified using Tau G2.1) using our extensions to Simmcast.

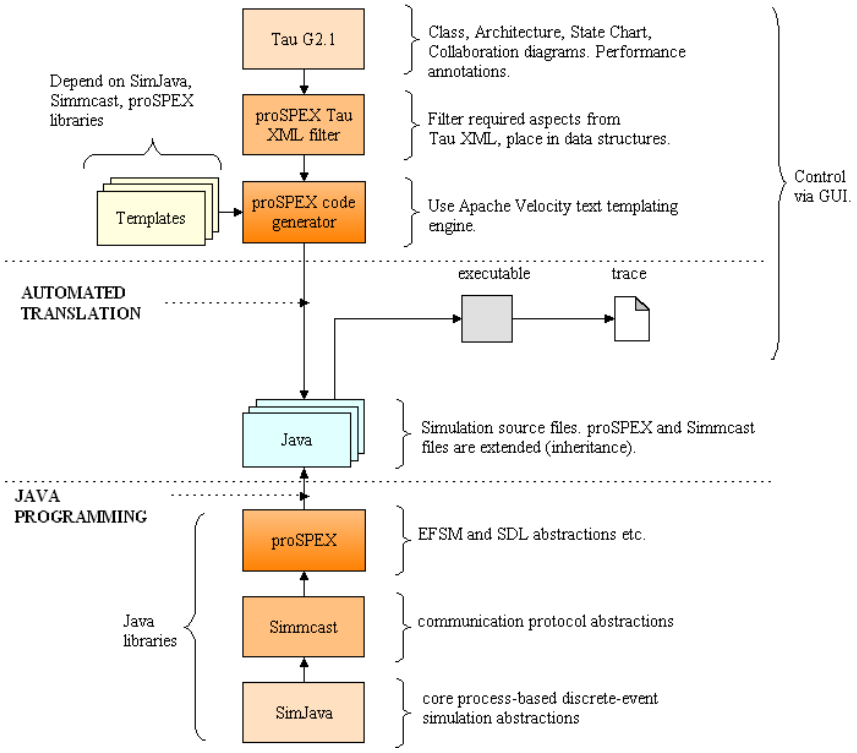

Figure 8: proSPEX overview

All of our extensions to Simmcast are contained in a *proSPEX Java package*.

Note that in extending Simmcast, we have effectively developed an *SDL runtime support system* as described by Mitschele-Thiel and Muller-Closterman in [MTMC99]:

> "The part [of the SDL runtime support system] typically provided by the implementer comprises the implementation or mapping of the primitives for interprocess communication, process management and timers on the respective primitives of the operating system [or virtual machine]".

6.2.1 Removal of the Simulation Description File

As motivated earlier in this chapter, we identified the need to replace the Simmcast simulation description file with pure Java method calls placed in a main file. We thus created a `ProNetwork` class which extends the Simmcast Network class by adding features used in setting simulation topology and startup parameters.

The primary attributes of class `ProNetwork` are a `ProArchitecture` object, network link settings (capacity, bandwidth, loss rate), a `SignalTargetHashTable` and other attributes. The system architecture is stored in the `ProArchitecture` object and is used in setting up paths between dynamically created nodes and daemon nodes. The `SignalTargetHashTable` is used in giving explicit addresses to signals, which had been sent using implicit addressing in the higher level state chart.

Most of the methods in the `ProNetwork` class exist to set up and manipulate the system architecture. The most important of these methods is `addSoftwareNode` which is used in overcoming one of the major Simmcast shortcomings; the inability to dynamically create active classes (or processes). The method `addSoftwareNode` is used in `ProNodeThread` instances (the protocol logic, contained in state charts in SDL, is placed in `ProNodeThread` instances with proSPEX) when a process spawns a child process.

6.2.2 Architecture Representation and Specification

As has been mentioned, we need a means of encoding the system architecture when dynamically creating nodes, which represent active classes. With Simmcast the topology was specified in the simulation description file however in the proSPEX extension the architecture would be specified in the simulation main file. The proSPEX code generator would

generate Java code which represents the system architecture using data structures created for the purpose.

With the proSPEX extension to Simmcast, the architecture is specified in two steps. The first is to add `NodeType` objects to a vector contained in a `ProNetwork` instance. The call to add a `NodeType` to the `ProNetwork` vector takes as parameters the node type (or class) and whether instances of the type in question are daemon or not. The second step in the specification of the architecture is to specify whether links between nodes exist. The existance of a uni-directional link in the architecture merely specifies that the possibility exists that node A may send packets to node B. The described information is stored in a `ProArchitecture` instance which contains a *path table* which is stored as a path matrix (or two-dimensional array).

An example of the specification of `NodeType` instances and network architecture links is given below. Note that Session nodes are all specified as being non-daemon (since the second parameter is *false* in the `addNetworkArchLink` call) and hence are dynamically created.

```
ProNetwork network = new ProNetwork("EsroInvoke");
...
network.addNetworkArchNode("InvokerUser",true);
network.addNetworkArchNode("Manager",true);
network.addNetworkArchNode("RoutingPeerProxy",true);
network.addNetworkArchNode("Session",false);

network.addNetworkArchLink("InvokerUser","Manager");
network.addNetworkArchLink("Manager","InvokerUser");
```

The architecture is used when one node dynamically creates another in protocol logic. In the ESRO protocol, a Manager process[1] would create a Session process upon receiving a service request. Once the Session process had been created the `ProArchitecture` path matrix is traversed and paths to and from the newly created Session process are created.

[1]Note that *process*, *active class* and *node* all have the same meaning; an instance with its own thread of control.

6.2.3 SDL Pid Expression Representation and Implicit Addressing Representation

SDL Pid expressions (*self*, *sender*, *parent* and *offspring*) which are mainly used for signal addressing as well as implicit signal addressing, are high level facilities that must be mapped to low level representations. In the case of mapping to an implementation these values may be mapped to IP addresses and port numbers. In our case, we must map the values returned by such expressions to integers values (each node in Simmcast has an integer address).

Maintaining values for the mentioned Pid expressions is done by including and maintaining a set of *implicit SDL state machine variables* in the state machine protocol logic that is generated by the proSPEX code generator. For example, a `sender` integer variable is included in each generated state machine. This variable is set each time a packet is received by the state machine. Likewise, if a node spawns child nodes, an *offspring* variable may have to be included as a variable and set once the child had been created. The *offspring* variable is only included if the parent node communicates with its child nodes.

In the case of implicit addressing, which is regularly used in SDL state machines, a `getSignalNetIdTarget` method is called prior to the signal being sent. We have implemented this method in the `ProNode` class, which extends the Simmcast `Node` class. An example of the use of the `getSignalNetIdTarget` method in protocol logic is given below. We can see that prior to sending a SAPsaturated packet, the network ID (a variable named *destination* is used in the example) of the target node is determined using the `getSignalNetIdTarget` method.

```
   int destination = this.myself.getSignalNetIdTarget("SAPsaturated");
Packet sendMe = new Packet(source, destination,...);
send(sendMe);
```

The `getSignalNetIdTarget` method relies on an additional data structure that is populated as a part of the topology and setup parameter specification. A hash table is used to map from a signal name (the signal name is the key) to a vector of target node types. The reason why a vector is returned is because it is possible for a signal to be sent to multiple types of nodes. Each node has a set of neighbouring nodes, that is, nodes to which it has paths, and this set is stored as a node member variable. What the `getSignalNetIdTarget` method does is to find the **single** match between the vector of target node types returned

by from the hash table and the neigbouring nodes. Once the match has been found the network ID of the node in question is returned. If multiple matches were found the SDL specification is incorrect since with implicit addressing the assumption is that each signal has a unique target.

A proSPEX class `SignalTargetHashTable`, is used in the `getSignalNetIdTarget` method. The hash table is populated by looking at the *required* interfaces specified in the UML 2.0 specification for each active class. The signals specified in these interfaces are all the signals that the active class in question can receive. So, for each such signal its name and the node type of the target is added to the `SignalTargetHashTable`. A code sample illustrating the population of the `SignalTargetHashTable` in the case of a Manager active class is given below.

```
/**
* Target: Manager
*/
// Interface: I_UserToMan
network.sigTargetHashT.addSignalTarget("invokeRequest","esro_invoke.Manager");
// Interface: I_RPProxyToMan
network.sigTargetHashT.addSignalTarget("invokePDU","esro_invoke.Manager");
network.sigTargetHashT.addSignalTarget("segInvokePDU","esro_invoke.Manager");
// Interface: I_SessToMan
network.sigTargetHashT.addSignalTarget("rel_Inv_RefNu","esro_invoke.Manager");
network.sigTargetHashT.addSignalTarget("rel_Inv_ID","esro_invoke.Manager");
```

6.2.4 Finite State Machine Representation

In representing FSMs we had the choice of either using case statements, the State Pattern [vBB99] or the *van Gurp approach* [vBB99]. We used simple case statements to implement state machines. This is because in our case FSMs will be generated by a code generator and not by a developer. Hence the problems of FSM evolution, instantiation and data management, which are addressed by other approaches, do not apply.

We give an example of the core parts of a proSPEX state machine. We translated from a Telelogic Tau FSM model of the ESRO[2] Invoke Service to an equivalent state machine

[2]An overview of the ESRO protocol is given in Appendix C and the protocol is also used in our performance analysis case-study in Chapter 7.

model. By doing so we determined the design of the templates of the proSPEX code generator.

The first part of the state machine is a set of variables which are not all explicitly contained in the associated Tau state machines. The sets of variables contained in a state machine are shown in the code sample of class `SessionThread` directly below.

```
class SessionThread extends NodeThread {

   /*********************************************
    *
    * STATE MACHINE CONTROL VARIABLES [1]
    *
    *********************************************/

    private int state = 0;
    private int previous_state = 0;
    private int event = 0;

    private static final int STATE_Start_STA01 = 0;
    private static final int STATE_InvPDUSend_STA2 = 1;
    ...

    private static final int EVENT0_invokeReq = 0;
    private static final int EVENT1_retransTime = 1;
    private static final int EVENT2_segRetransTime = 2;
    ...

    boolean done = false;

    /*****************************************************
    *
    * EXPLICIT STATE MACHINE VARIABLES (FROM UML 2.0 SPEC) [2]
    *
    *****************************************************/
```

```
InvokeSDU invSDU;
int invokeRefNumb;
InvokeProvConfSDU invPConf = new InvokeProvConfSDU();
int CLRO_SMALL_SDU_MAX_SIZE = 1500;
...

/*******************************************************
 *
 * IMPLICIT STATE MACHINE VARIABLES (FROM SDL) [3]
 *
 *******************************************************/

 int source = 0; // our own address used when creating packets

 int sender = 0; // the address of the node/process that sent the last signal

 /*******************************************************
  *
  * IMPLICIT STATE MACHINE VARIABLES (FROM DIAGRAM) [4]
  *
  *******************************************************/

  /* RECEIVING:
   **/
  PacketType invokeReq = new PacketType("invokeReq");
  PacketType invokReqPeer_Invoke_PDU = new PacketType("invokReqPeer_Invoke_PDU");
  ...

  /* SENDING:
   **/
  PacketType invokeProvConfirm = new PacketType("invokeProvConfirm");
  PacketType sendSegInvPDU = new PacketType("sendSegInvPDU");
```

```
    ...

/*********************************************
 * VARIABLE END
 *********************************************/
```

There are five sets of variables in proSPEX FSMs. The first set of variables are those that are used in the state machine execution control. These variables store the current state, previous state and current event being processed. In addition, each state and event that can be processed is mapped to a sequential integer value. The second set of variables are those that are defined by the user in the UML 2.0 specification, while the third are implicit SDL state machine variables. The implicit state machine vairables are used to represent the current SDL Pid expression values. Note that in the example only the *sender* value is stored since the other expressions were not used in the FSM in question. The final set of state machine variables are the implicit state machine variables that are derived from the active class's provided and required interfaces. Due to the design of Simmcast each signal that can be sent and received by the FSM has to have an associated `PacketType` object which takes the signal name as a parameter.

What the example of the four different sets of variables illustrates, is that only a small subset of the FSM variables are specified by the user, the majority are *implicit* variables that are to be generated by the proSPEX code generator.

Each FSM has a control section and in the following example we see how it is implemented. Each FSM has its control specified in an `execute()` method, which is the method that is called by the simulation scheduler when a `NodeThread`, or process, executes for the first time.

```
public void execute() {

// our own address used when creating packets
source = myself.getNetworkId();

while (done == false) {

    try {
```

```
switch (state) {

case STATE_Start_STA01: {

    // determine what the next input signal/packet is....
    Event event = getInputSignal();

    switch(event.getType()) {

        case EVENT0_invokeReq:
            previous_state = state;
            state = processEvent0_invokeReq(event);
            break;

        case EVENT4_invokReqPeer_Invoke_PDU:
            previous_state = state;
            state = processEvent4_invok(event);
            break;

            default:
            tracer.signal_discard(event.getPacket(),this.state);
    }
}
break;

case STATE_InvPDUSend_STA2: {
   ...
}
break;

default: {
    throw new StateMachineException("There is no default state.");
```

```
        }
    } // switch

    /* Here we have a successful transition from one state to another.
    * We record this in the trace.
    **/
    tracer.state_transition(this.myself,previous_state,state);

} // while(true) }
```

The example shows that we have implemented the state machine procedurally using an outer `while` loop and nested switch statements. Once in the `while` loop the first switch statement uses the `state` control variable to switch to the correct case statement. The proSPEX filter and code generator store the initial state of the UML 2.0 specification in order to set the initial value of the `state` variable to the correct state.

After having reached the correct case statement, and hence state, different "transitions" could be traversed depending on the input signal received. We call these input signals *events* in the state machine example and each "transition" is mapped to an event variable and associated event processing function. In each state the first thing that happens is that the `event` variable is set by a call to the `getInputSignal()` method as follows: `Event event = getInputSignal()`. This is a blocking call, meaning that the execution of the FSM is suspended by the simulation scheduler untill some signal arrives at the FSM. Once a signal arrives the execution of the FSM continues and another switch statement is executed resulting in a case statement being executed as can be seen in the example below.

```
case EVENT0_invokeReq:
    previous_state = state;
    state = processEvent0_invokeReq(event);
    break;
```

The first thing that happens in all event processing case statements is the setting of the `previous_state` variable which exists for tracing purposes. The next step is the execution of the associated event processing function `processEvent0_invokeReq(event)` which, apart from executing all actions in the associated FSM transition, returns and sets the `state`

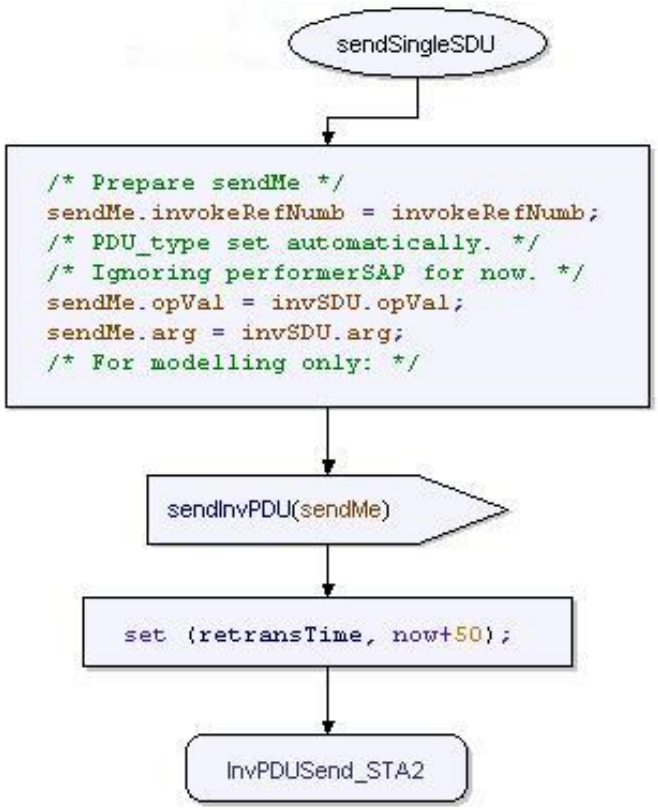

Figure 9: An example of a junction symbol

variable. Finally, a `break` statement is executed and control passes to the outer state based switch statement. Directly before this happens a call to the simulation tracer is made; `tracer.state_transition(this.myself,previous_state,state)`. This results in a state transition from state `previous_state` to `state` being recorded on the simulation trace.

The next part of the state machine implementation that we discuss are the individual event processing functions which generally represent entire transitions and sometimes parts of transitions. Cases in which event processing functions represent parts of transitions are when junction symbols are encountered. Junction symbols are ellipses which are used to segment transitions. An example of a junction, labelled `sendSingleSDU`, is shown in Figure 9.

Another situation where we have found the need to generate event processing functions in the proSPEX code generator is in the case of loops that are specified graphically, an

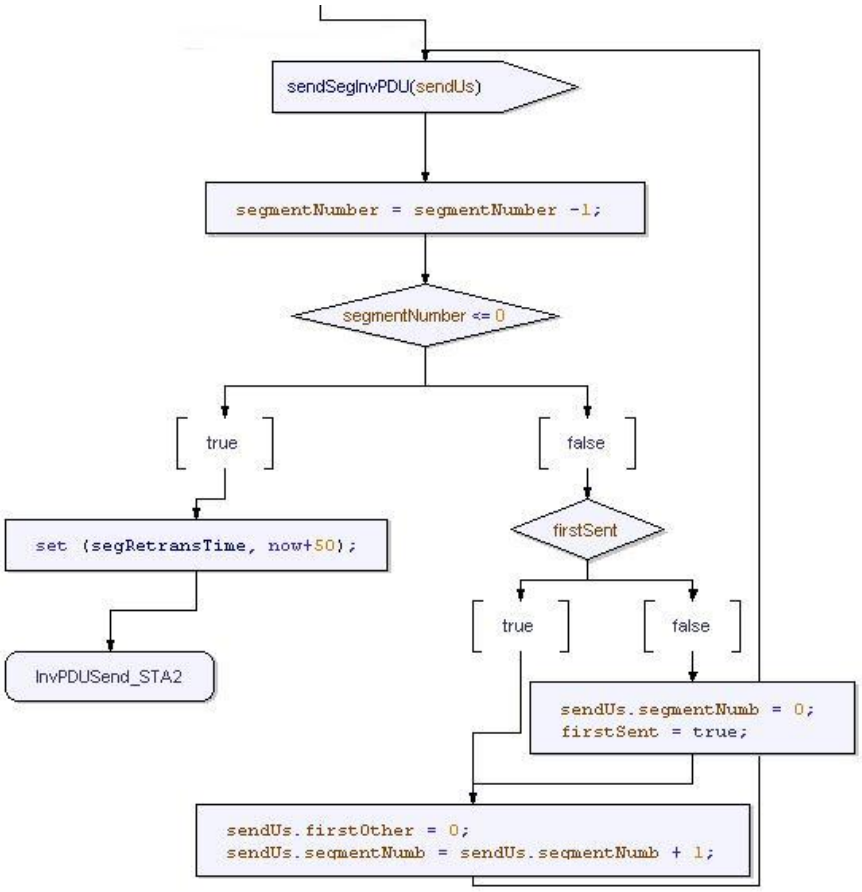

Figure 10: An example of a graphical loop

example of which is given in Figure 10. Such loops may be easy to detect by inspection but less so algorithmically when parsing the Tau XML. One way in which a parser could detect a *graphical loop* is when a flow line connects with another flowline which had already been visited, this would be a clear indication of a loop regardless of whether a decision symbol is present in the transition. A problem we have is in the choice of what type of loop we would we generate in the case of a graphical loop, would it be a `for`, `while` or `do-while` loop? In addition, how would our parser know which decision symbol is the one controlling the number of loop iterations? One solution to this problem is simply to create a function that is called each time one flow line meets another which we have already visited in travelling down the XML tree that represents the state chart model. In effect we treat the meeting point of two flow lines as if the user had placed a junction symbol at that point. The

proSPEX filter and code generator components must thus be able to detect the meeting of flowlines.

We have covered a subset of the implementation aspects of proSPEX FSMs. From this subset we hoped to show that developing the FSM representation was non-trivial, in fact a substantial part of our development time was devoted to the task. We will re-visit some of these aspects, and investigate others, when the proSPEX text templates are described in Section 6.3.

6.2.5 Additional Trace Events

A set of trace messages, listed in Section 5.2, are required to generate the performance metrics we are interested. Adding the trace routines to Simmcast was easy due to Simmcast being specifically designed to be extended.

6.3 Translating from Tau XML to proSPEX

Having extended Simmcast with the features that we require in order to generate an equivalent simulation model, our next task was to automate the process of translating from the Tau model to a simulation model. We have used a tier-generation process, in which an entire program (or *tier*) is generated using a text-templating engine, as described by Jack Herrington in [Her03]. We use the Jakarta Velocity text templating engine (TTE), as is done in the code generator of the Poseidon UML modelling tool by Gentleware AG.

6.3.1 A Code Generation Example

We use a simple example to illustrate the use of a TTE and also the steps that are used in developing the code generator in our context. Our goal in this example is to generate the Java classes that represent all classes that are used as the signal parameters in required and provided interfaces. Figure 11 shows the architecture diagram of the `RoutingPeerProxy` class. We presume that proSPEX had filtered the Tau XML and detected that the classes used as parameters in the signals contained in the `I_PeerToPeer` interface would have to be generated. If we look at interface `I_PeerToPeer`, we see that `Invoke_PDU`, `Seg_Invoke_PDU` and other classes are used as parameters to the `PeerDatagram` signal. proSPEX would have to find each such class in the Tau XML and then determine what types of relationships each class is involved in. We can see that the classes are involved in an inheritance hierarchy

with class PDU being at the top of the hierarchy. Thus, proSPEX must navigate the Tau XML and find the parent classes of each child class.

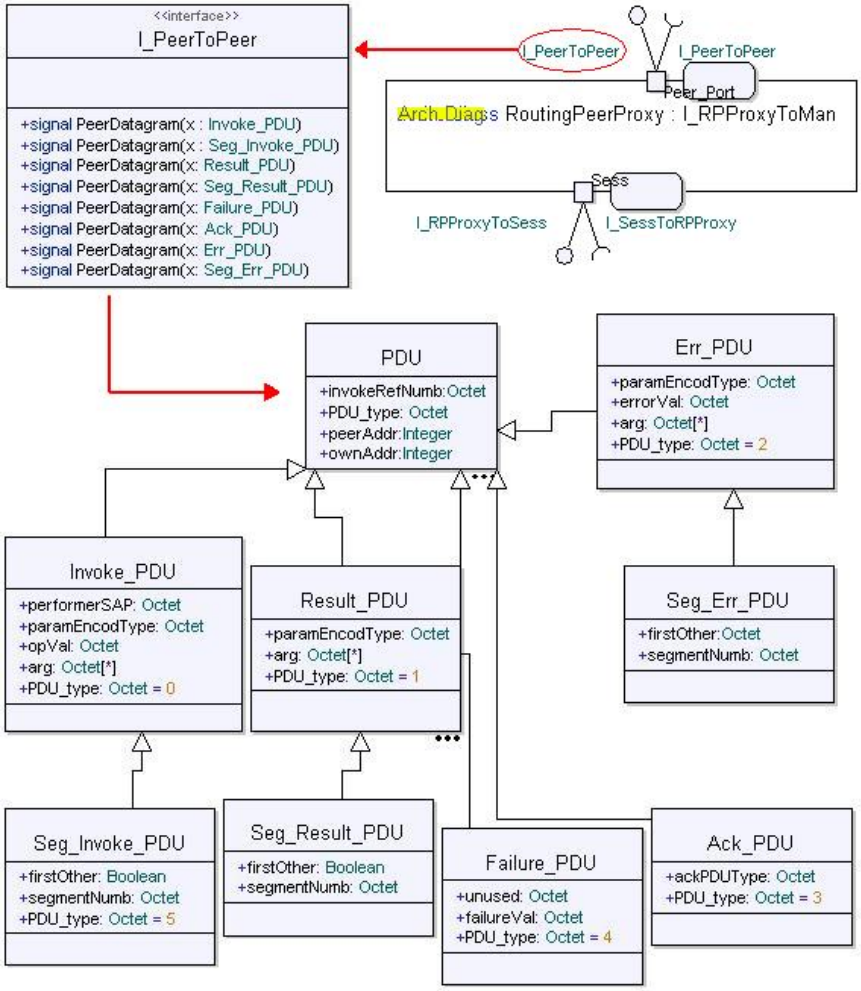

Figure 11: An example of signal parameters

Having established what proSPEX must do, we inspect the Tau XML and determine where the required information is located, as is shown in Figure 12. We use an XML viewing tool, XPath Explorer, to located one of the child classes and also the elements that indicate that the class in question is involved in an inheritance hierarchy. With approximately 11 000 verbose lines of Tau XML that represent our model, this inspection process is time consuming. The intuitive nature of XML however helps us in finding the required elements. We then use a combination of XPath queries derived from XPath Explorer (queries which

can be used to search in XML) and JDOM (a Java XML processing library) tree navigation to find each `PDU` class shown in Figure 11. As each class in encountered we firstly use regular expressions to extract the desired text from the class and then add a Java object representing the discovered class to a data structure, such as a linked list.

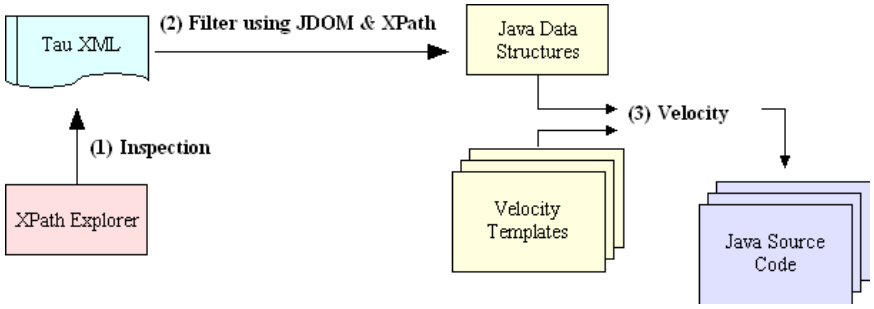

Figure 12: The Code Generation Process with Velocity

The final step in the code generation process is to use the Velocity TTE to generate the Java class files. Each Velocity template file is written using the Velocity Template Language (VTL) [Pro04] which provides a simple way of incorporting dynamic content in pre-written text. VTL uses variable references that can refer to something defined in Java code, such as a data structure. As an example in the sample Java code shown below, we see a `HashSet` data structure being populated with `SignalPayload` objects via a call to `getAllSignalClasses`. We iterate through the list of signal parameter classes and place them in a Velocity context. All that this means is that we give the object a variable name, in this case `signalPayload`, that will result in all of this object's methods being available in the template.

```
HashSet signalParamClasses = sigParamGen.getAllSignalClasses();

// Iterate through all HashSet entries...
SignalPayload sigPayload = (SignalPayload)iter.next();

// make the sigPayload object accessible in the template...
context.put("signalPayload",sigPayload);
```

In the sample VTL code shown below, we see how each `signalPayload` object is accessed in the template. For example, the code `$signalPayload.ClassName` will result in the class

name being substituted in the pre-written text.

```
class ${signalPayload.ClassName} #if ( $signalPayload.Parent != ""
) extends $signalPayload.Parent #end {

    #foreach ( $attrib in $signalPayload.Attribs )
    public $attrib.AttribType $attrib.Name;
    #end

    #if ($signalPayload.Networked)
    public int length;
    #end

    public ${signalPayload.ClassName}() {

    #foreach ( $attrib in $signalPayload.Attribs )
    #if ( $attrib.DefaultValue != "" )
    $attrib.Name = $attrib.DefaultValue;
    #end
    #end

    }
```

6.3.2 The proSPEX Tau Filter

The proSPEX Tau Filter is a set of classes that are responsible for placing required Tau XML data into objects and data structures that can be readily accessed using Velocity templates and VTL. In doing so the Tau Filter makes certain assumptions regarding the model that it is processing. For example, the Filter expects one and only one collaboration diagram depicting the simulation scenario. In addition, the Filter assumes that all of the classes in the scenario are daemon classes that exists at system start-up time.

The proSPEX Filter is divided into classes `MainXMLFilter`, `SignalXMLFilter` and `StateMachineXMLFilter`. The `MainXMLFilter` is responsible for gathering the data that is contained in the simulation main file. This includes the network link characteristics that

are specified in a comment symbol in the simulation main file, the child active classes (or parts) of the daemon classes and the signals contained in the required and provided interfaces. The `SignalXMLFilter` is responsible for placing all the signal payload class data into `SignalPayload` classes. Lastly, the `StateMachineXMLFilter` is responsible for filtering and then placing all state machine data.

These classes are naturally supported by a host of helper classes such as `ProSPEXTypes`. Class `ProSPEXTypes` is used to convert from Tau UML basic types to Java types using an XML file containing the mappings. For example both `Pid` and `offspring` are mapped to `int`. Other helper classes are used to contain model information that is eventually used in the templates. Such classes include `Attrib`, `Signal`, `SignalPayload`, `State`, `StateMachineData`, `Transition` and `SystemData`.

6.3.3 The proSPEX Code Generator

As we have seen in the earlier example, the Velocity templates and the various XML filter classes in combination form the proSPEX Code Generator. We have created four templates, given in Appendix D, that are used in the code generation process:

- **main.vm**: this is the simulation main file template in which the network path characteristics, ProNetwork, Nodes, paths between nodes and architecture is specified.
- **node.vm**: each UML 2.0 active class is mapped to a Node and this template is responsible for generating the Node classes.
- **signalparam.vm**: this template is responsible for generating all classes that are used as signal parameters.
- **statemachine.vm**: the various parts of FSMs, implemented as discussed in Section 6.2, are generated using this template.

Creating the above mentioned templates was significantly less time consuming than developing the XML filters and classes used to package the data that is required in the templates. In fact the manual part of the code generation process shown in Figure 12 resulted in the development of the XML filters being the most time consuming aspect in this dissertation. This was largely because we had to determine the meaning of thousands of lines of XML by inspection due to a lack of supporting documentation (such as a DTD or other Telelogic documentation).

6.3.4 Graphical User Interface

We created a rudimentary GUI that can be used import UML 2.0 protocol specifications, generate the simulation code and run simulations. The GUI consists of two windows, one which displays the simulation trace and one which displays simulation results, as is shown in Figure 13. We mainly used the GUI to control our case study experiments which are detailed in the next chapter. We used Excel, which expects data in a comma seperated values (CSV) format, to plot results.

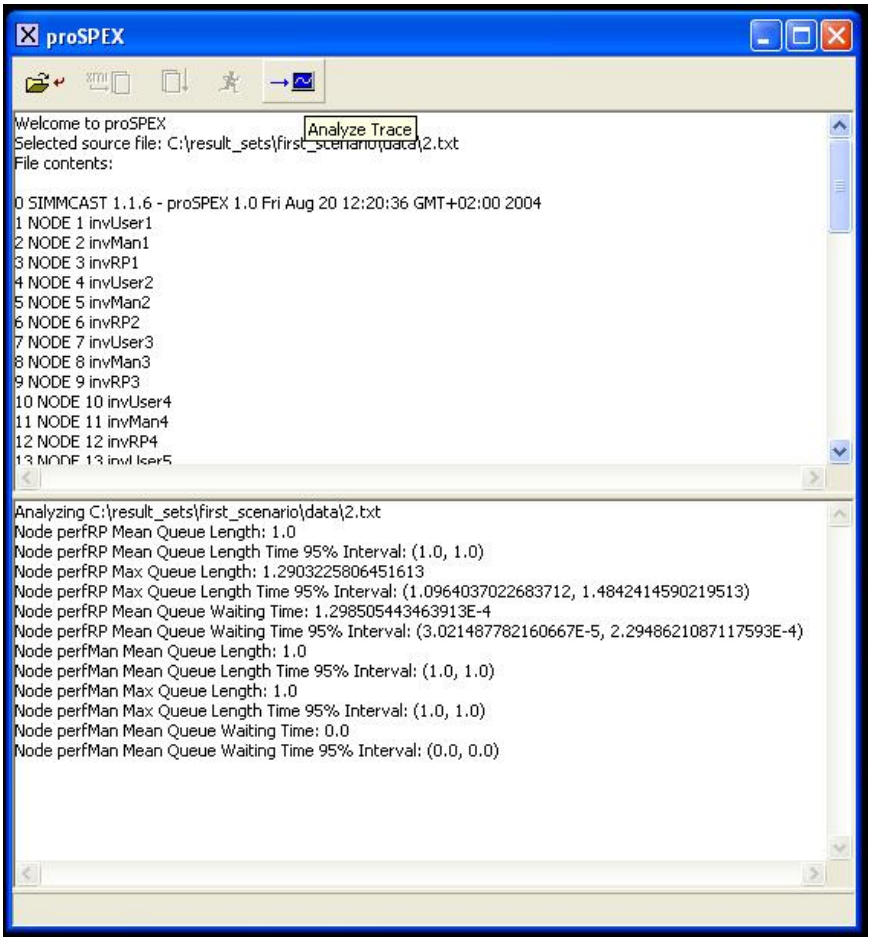

Figure 13: The proSPEX GUI

6.4 Summary and Conclusion

We have seen that proSPEX is comprised of the proSPEX extension to Simmcast, which is in essence a SDL runtime support system, the proSPEX Tau filter and lastly the proSPEX code generator with its associated templates. Although we have shown that we can successfully generate simulation programs we have found the most challenging aspect the filtering of the Tau XML in which we had to determine the meaning of many thousand lines of XML by inspection due to a lack of supporting documentation. This development process is both slow and error prone and should only be attempted when processing small subsets of models, such as class diagrams. In our case, we have attempted to process class, architecture, state chart and collaboration diagrams which in retrospect was overly ambitious given the technical limitations that we have explained.

Chapter 7

Performance Analysis Case-Study

7.1 Introduction

In this chapter we use the ESRO protocol in a performance evaluation case-study. We demonstrate the utility of proSPEX by determining a set of performance statistics (with appropriate confidence intervals) that are useful in the context of the case-study. Our intention is to apply the proSPEX methodology and approach to performance evaluation to a typical scenario. The intention of this case-study is only to illustrate the utility of proSPEX and is not a performance study in the first case.

7.2 Experiment Specification

Our goal is to investigate the performance of ESRO in a typical system scenario. Not only do we need a good representation of the system, and hence realistic parameter values, but also a good representation of the workload of the system.

Workload characterisation is outside the scope of this dissertation. However, the parameter values we have chosen we believe plausible since they are based on analytic models[GB04] and practical experimental results[CP03][RCP02] where possible.

Before detailing a particular experimental scenario (with its associated parameters and workload) we briefly mention the primary features of the Efficient Short Remote Operations (ESRO) protocol (see Appendix C for a detailed discussion). The service ESRO offers is a reliable connectionless transport for wireless links when efficiency is of concern. The service supports applications based on a remote operations model that is largely the same as the

Remote Procedure Call (RPC) model [Mic88]. The identification of the encoding mechanism in use (e.g. Abstract Syntax Notation One (ASN.1) and its Basic Encoding Rules (BER)) is supported. An ESRO user that invokes an operation is called an *invoker* whilst the ESRO user that performs the operation is called the *performer*. By default, operations are asynchronous and the invoker may invoke concurrent operations without waiting for a reply. Applications using ESRO typically include efficient short message delivery and submission, credit card authorization and white pages lookup.

7.2.1 Experiment Scenario: Wireless E-Commerce

We have specified the ESRO Invoke service using Telelogic Tau G2.1 and shown how to map from a Telelogic UML 2.0 model to a *proSPEX* simulation model using the proSPEX methodology and approach to performance evaluation.

In our scenario ESRO is used in a *credit card authorization application* in which a number of clients invoke operations on a single server acting as the ESRO provider. The clients are mobile devices such as cell phones and PDAs while the server is a high-end desktop machine. The wireless link between the clients and server is a GPRS link. Each client invokes a number of operations on the server with each service data unit (SDU) having a fixed size. Such a fixed message size is plausible in e-commerce applications which use the ISO 8583 standard [fS03] for communication between financial systems. Figure 14 shows our scenario in which an e-commerce user application (deployed on a cell phone) is connected to an e-commerce server via ESRO. The e-commerce server is linked with an EFT transaction switching gateway.

The primary aim of the performance analysis study is to determine whether the service data unit (SDU) throughput at the server is able to meet a required level. In our scenario, the required minimum SDU throughput at the server is taken to be equal to the maximum number of transactions per second that the EFT switching gateway is able to process. Hence if the EFT transaction switching gateway can process 20 transactions per second, the ESRO provider in the e-commerce gateway should be able to deliver a throughput rate of 20 invoke indications per second, with each invocation indication carrying a transaction in an SDU.

If we view the ESRO provider as an abstract system with job arrivals and completions, we would say that the requirement is for the throughput (or completion rate) to be equal to 20 jobs per second. In our experiments we want to ensure that the ESRO server is able

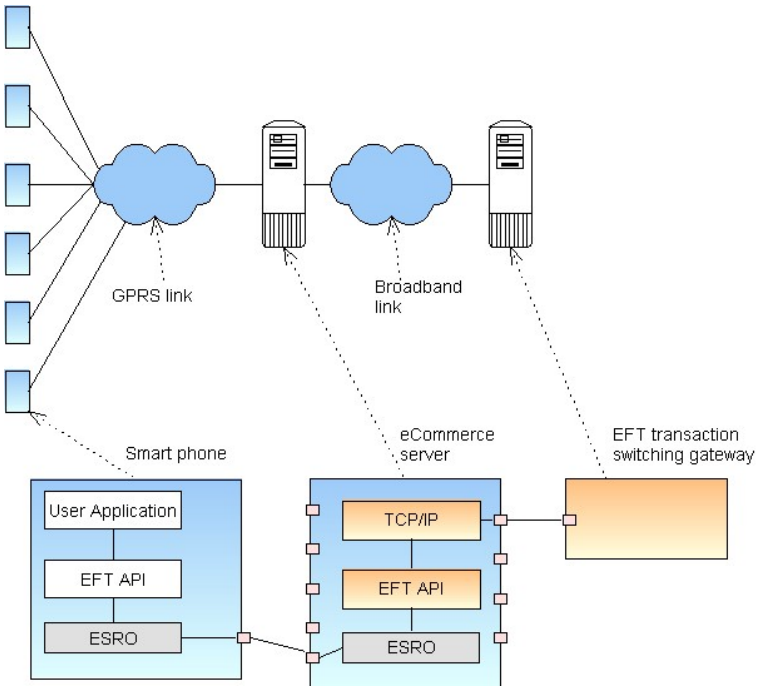

Figure 14: Experiment Scenario

to deliver a given invoke indication rate (or throughput).

If the required throughput cannot be reached performance statistics such as mean queue length, discarded signals and expired time-outs can be used to determine system bottlenecks. The result of analyzing the performance model would thus be a set of discrepancies between offered and required resource attributes [Gro02].

The parameters in our ESRO Invoke Service model are time-out settings, message size, network link characteristics, connector buffer sizes and nodal processing delay. In the next section we estimate suitable nodal processing delay parameters, network parameters and workload parameters. Once we have established suitable values for the parameters in the mentioned categories we can determine whether an ESRO provider will be able to meet the required service levels.

7.3 Model Parameters

In this section we estimate the processing, network and workload delay parameters used in our performance model. A summary of the parameters estimated in this section is provided in Section 7.4.

7.3.1 Processing Delay Parameters

It might seem that in the case of low bandwidth wireless links, processing delay would be insignificant when compared to network delay. This may be the case at each client; however, under heavy load processing delay could become significant at the server.

With Simmcast, a deterministic[1] or randomly distributed processing delay is set for each node. In order to do so we investigate the protocol processing that takes place and the protocol hardware environment.

In the case of a transport layer protocol (such as ESRO or TCP), the time taken to process a packet at an end-station or network node depends on protocol complexity, application code, processor power, I/O delay, network delays and context-switch delays. With a protocol such as ESRO the main source of delay are marshalling[2] delays, I/O delay, network

[1] A deterministic processing delay is probably a simplification of reality.

[2] When a method on a remote object is invoked, an RPC system marshals (writes and transmits) the method parameters, waits for the result of the method invocation and then unmarshals (reads) the return value[Mic04].

delays and context-switch delays. Since network delays are modelled we must estimate values that represent *marshalling delays*, *I/O costs* and *context-switch delays*. These delays are mainly dependent on the nodal architecture.

With nodal processing delay being dependent on a number of dynamic delays, which result from complex processes, it may seem that establishing nodal delay values is not feasible. However note that processing would take place on clients and servers where the *computational power* of the server would be at least an order of magnitude greater than that of the clients. The processing power is primarily a combination the CPU, RAM and storage at the node.

We consider the specifications of a high-end cell phone, the Sony Ericsson P900 which contains a 32-bit RISC ARM9 156 mHz processor and 48 MB RAM. Contemporary Internet server hardware may typically be dual Intel Pentium 4 Xeon 3.6 GHz CPUs, 2GB RAM, 10,000 RPM hard-disks with RAID 1 mirroring. From these details we could conclude that the average Internet server has computational power that is *one order of magnitude* greater than that of the average mobile terminal. We should therefore ensure that the processing delay we assign to nodes of the clients in our experimental scenario is an order of magnitude larger than that of nodes of the server.

A recent study [DMS03] compared the performance of ASN.1 BER and XML when transmitting data in an application used by health service professionals. ASN.1 message size ranges used in the study ranged from 235 bytes (0.2kB) with simple attributes and 1351 bytes (1.3kB) with complex attributes. The corresponding sender encoding times were 6.8ms and 10.5ms respectively while the recipient decoding times were 1.6ms and 3.5ms respectively. The measurements were taken using an IBM Java Virtual Machine, 650 MHz P3, 256 Mb RAM and Redhat Linux.

In order to use these results in our case-study, we firstly extrapolate to get processing times for a larger message size. In this process we simply multiply the message size, encoding time and decoding time by the fraction increase in each category when processing 235 byte and 1351 byte messages. The results of this extrapolation is shown in Table 1. After extrapolating we multiply each processing time by the proportional increase or decrease in processor MHz when using a 156 MHz and 3600 MHz processor to obtain the processing times listed in Table 2.

While we have estimated the ASN.1 coding and decoding times we still have to determine which nodes, in the simulation model, to assign the processing times to in the clients and

Size (byte)	Increase	Encoding (ms)	Increase	Decoding (ms)	Increase
235	5.7	6.8	1.5	1.6	2.2
1351		10.5		3.5	
1351	5.7	10.5	1.5	3.5	2.2
7754		**16.2**		**7.6**	

Table 1: Extrapolated 7754 byte Message Processing Times using ASN.1

Device	Encoding (ms)	Decoding (ms)
Smart phone	67.5	31.4
Internet server	2.9	1.4

Table 2: Scaled 7754 byte Message Processing Times using ASN.1

server. ESRO has Manager, Session, Routing Peer Proxy, Invoker User and Performer User nodes. The encoding takes place when the Manager receives an SDU from the Invoker User, we therefore set an encoding (or *send time*) of 67.5 ms for each Manager node in each client. In the server the decoding will take place directly before the SDU is passed to the Performer User, we thus set a *receive time* of 2.9 ms in the Performer User.

7.3.2 Network Parameters

Here we use empirical results obtained from Vodafone UKs GPRS network ([CP03][RCP02]) to parameterise the GPRS network link in our experimental scenario. The network parameter values we need are the loss probability, bandwidth, delay distribution and delay distribution parameters.

Assuming that CS-2 encoding[3] is used and a 3+1 mobile terminal (3 downlink, 1 uplink channel), the theoretical maximum downlink and uplink bandwidth is 40.2 kbit/s and 13.4 kbit/s respectively. The observed maximum bandwidth was 33.2 kbit/s and 11.2 kbit/s respectively. Delay distribution functions are not provided in [CP03][RCP02]; instead, histograms are provided showing uplink and downlink delay distributions in the case of sending one thousand 64 byte UDP[4] packets. A round trip (RTT) delay of approximately

[3]With GPRS the transfer speed depends the number of downlink and uplink channels as well as the channel encoding used. The best encoding is CS-4 while the worst is CS-1. For example the maximum bandwidth per time slot, when using CS-4 and CS-1 is 21.4 kbit/s and 9.05 kbit/s respectively.

[4]Note that the results given in [CP03][RCP02] are from transport layer measurements.

a second was found to be common. Since we are modelling the ESRO Invoke Service, we are interested in the uplink (mobile station to server) characteristics. The uplink latency range was between 400 and 1300 ms. Packet loss probability is described as rare and hard to quantify in [CP03], although analytic models have shown [GB04] a range of 0 to 0.150.

Using the results in [CP03][RCP02][GB04] we therefore conclude that we have a maximum uplink bandwidth of 11.2 kbit/s (1400 byte/s), a delay of between 400 and 1300 ms and a loss probability of between 0 and 0.150.

7.3.3 Workload Parameters

From the experimental scenario we have established that the ESRO server should be able to deliver a given throughput rate (or invoke indications per second received at the server). In our first experiment we use a rate of approximately 1 invoke indication per second. In order to achieve 1 invoke indications per second we use 5 clients, each sending an invoke request to the ESRO system with an inter-request time of 800 ms. If we are to run the simulation for approximately 90 simulated seconds[5] each client sends 113 method invocation requests in that period.

In order to avoid having each client send an invoke request at exactly the same time, each client starts sending invoke requests after a random delay between 0 and 1 second. We are interested in the invoke indication throughput at the ESRO server and whether resource saturation occurs.

7.4 Parameter Summary

The following is a summary of the ESRO system requirements and model parameters that we established in Sections 7.2.1 and 7.3.

Parameters that are needed for performance evaluation include the QoS parameters, and a description of the workload intensity:

- The number of active users in the system is 5.
- Each active user sends invoke indications with an interval of 800 ms for a period of 90 s.

[5]With an observation window of 90 s we have a *terminating* simulation that is said to have a *cold-start* meaning that the initial period in which the system is empty is included in the observation period.

- Each active user sends the first invocation request after a random zero to one second delay.
- (requirement) The ESRO server invoke indication throughput should reach 1 per second without resource saturation.

We have the following parameter values (they are labelled measured, estimated, or assumed):

- (assumed) A fixed message size of 5kB is used.
- (assumed) The maximum ESRO protocol data unit size is 1500 bytes.
- (estimated) For a cell phone the ASN.1 encoding and decoding time of a 5kB byte message is 67.5 ms and 31.4 ms respectively.
- (estimated) For an internet server the ASN.1 encoding and decoding time of a 5kB byte message is 2.9 ms and 1.4 ms respectively.
- (assumed) With the ESRO Invoke Service PDU packets are sent from the invoker to the performer, we set the *send time*[6] for the cell phone clients at 67.6 ms and the *receive time* to be 2.9 ms at the ESRO server.
- (assumed) In the case of the clients the send time of 67.6 ms is set for each Manager while for the server the receive time of 2.9 ms is set for the Performer User. All other nodes in the network have the default send and receive time of 0ms.
- (assumed) The smart phones use CS-2 encoding and are '3+1' mobile terminals.
- (estimated) For the GPRS link (which is modelled at the ISO network layer) we have a maximum uplink bandwidth of 11.2 kbit/s (1400 byte/s), a delay of between 400 to 1300 ms and a loss probability of 0.01.
- (assumed) The queue size assigned to connectors in our model is infinite[7]

[6] In our examination of Simmcast, Chapter 8, we saw that each Node can have a send time and/or receive time assigned in order to model processing delay.

[7] In this case we have to look out for monotonic queue size increase in the ESRO server.

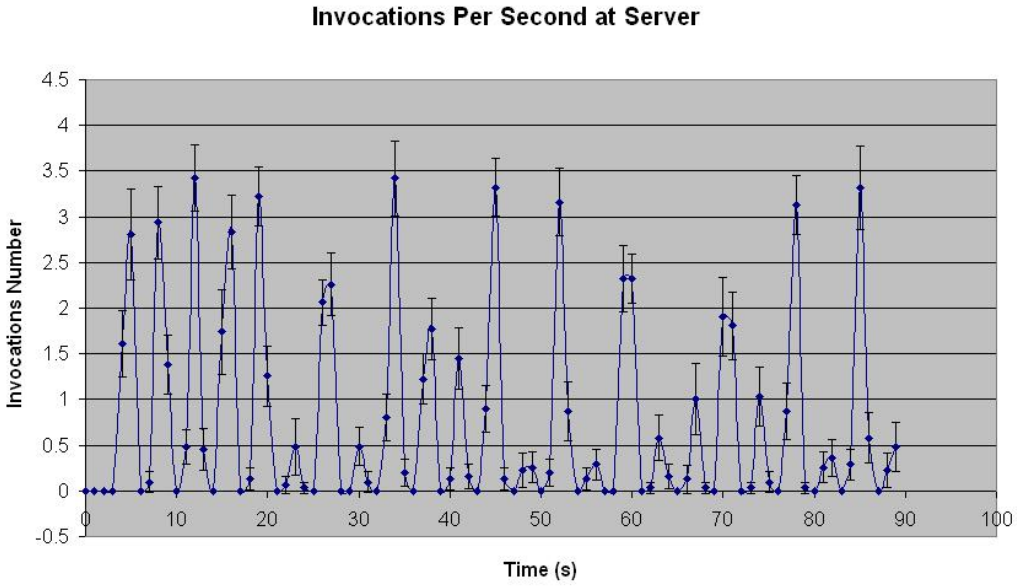

Figure 15: Invocation Throughput at the Server modelled by proSPEX

7.5 The Experiments

Using the parameters and workload summarized in Section 7.4 we conducted a number of simulation runs[8] and plot the results. Figure 15 illustrates the throughput (invocations per second) received at the server while Figure 16 illustrates the cumulative invocations received at the server.

Figure 15 shows that the server was able to meet the required throughput rate. This is clear from Figure 16 since the gradient shows no sign of decreasing. If the gradient were to decrease saturation at the ESRO server would be a likely cause.

Having established that the ESRO server can meet the required throughput we investigated queue length and waiting time metrics. We determined further performance metric values since *system tuning* (determining the optimal value of parameters) may still be possible despite the requirements being met. We focus on the mean and maximum queue lengths and mean queue waiting time at daemon nodes of the server.

[8]Experiments were repeated over thirty times for a high level of confidence in results. In addition 95% confidence intervals are used unless otherwise stated.

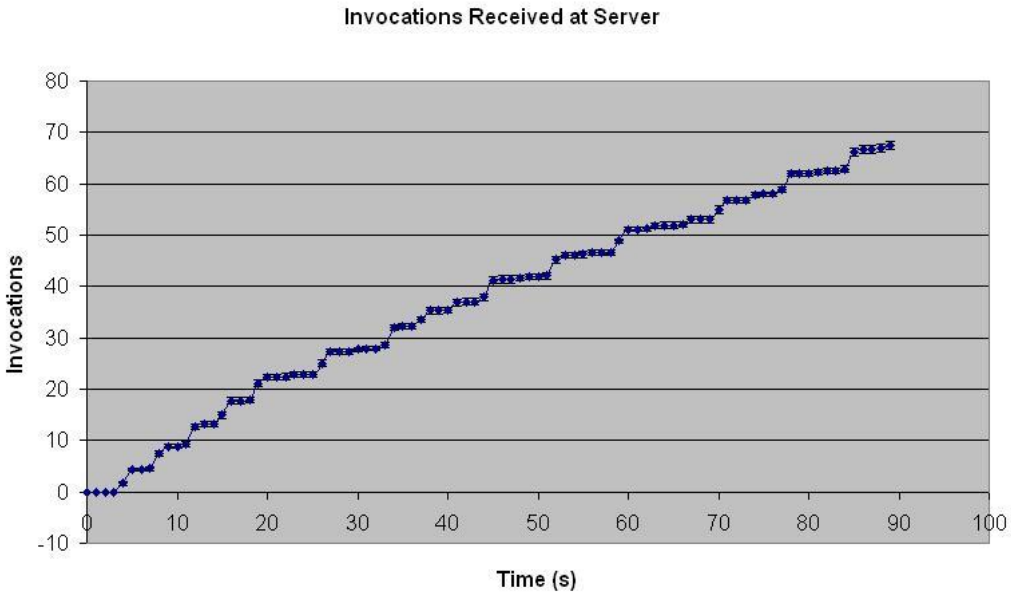

Figure 16: Cumulative Invocations at the Server modelled by proSPEX

We modified our model in order to increase the load on the server. We first made the assumption that the processing power of the server is only double of that of the clients and each node in the model has a processing delay set which honours this assumption. We also doubled the number of clients in order to increase the load on the server. With these changes we repeated the experiment and observed mean and maximum queue lengths in the server and mean queue waiting times in the server. The resulting queue lengths in the daemon nodes of the server (the Manager and Routing Peer Proxy nodes) are shown in Table 3.

In Chapter 5 we saw that the mean queue waiting time (or the average time that a signal spends in the queue of a process) shows whether a process handles messages fast enough. In addition, a high mean queue waiting time indicates many retransmission messages in a process queue due to the timeout of the sending process being too short. Table 3 shows relatively low mean queue waiting times at the Manager and Routing Peer Proxy nodes. The queue lengths, which model the buffers of a communication system, show the expected buffer size required by the system. With a maximum queue length of 7 packets at the

	Manager Node	Proxy Node
Mean Queue Length 95% Confidence Interval	1.00 (1.00 - 1.00)	1.33 (0.79 - 1.88)
Max Queue Length 95% Confidence Interval	1.00 (1.00 - 1.00)	7.00 (6.34 - 7.66)
Mean Queue Waiting Time 95% Confidence Interval	1.24^{-5} (0.0 - 4.45^{-5})	6.95^{-2} (5.21^{-2} - 8.69^{-2})

Table 3: Queue Based Performance Metrics for ESRO Server Daemon Nodes

Routing Peer Proxy, the largest buffer required in the ESRO server is 10.3 kB. The queuing metrics we have discussed show that the ESRO server is able to process its load without requiring significant system resources. This is partly due to the simplicity of the protocol, but it could also be indicative of the simplifying assumptions and simplicity of our model.

As a final experiment we evaluate the throughput of the system under varying connector reliabilities and varying timeout rates. We measure the average invoke indication rate (or throughput) at the server with 5 different network link loss probability values (0, 0.05, 0.1, 0.15, 0.2) and three different service data unit (SDU) retransmission timeout settings. The results of the experiment, as shown in Figure 17, shows the expected reduction in throughput for lower connector reliabilities. The vertical axis is connector throughput and the horizontal axis is the packet loss probability. The graph shows that the fast timeout is the worst performer, this is due to the sender making a retransmission before all of the previous PDUs of the SDU arriving at the receiver. We also see that at the slow timeout setting is superior to the medium setting at low (0 to 5 percent) error rates, while the latter is superior at higher (6 to 18 percent) error rates.

7.6 Conclusion

We have demonstrated the utility[9] of proSPEX by determining a set of performance statistics of our experimental scenario. This scenario was the use of ESRO as the transport layer protocol in an a mobile e-commerce application. We parameterized the model with time-out settings, a service data unit message size, network link characteristics, connector buffer sizes and nodal processing delay settings. The goal of the experiments was both to

[9]The purpose of this case-study was purely for demonstration purposes and not an end in itself.

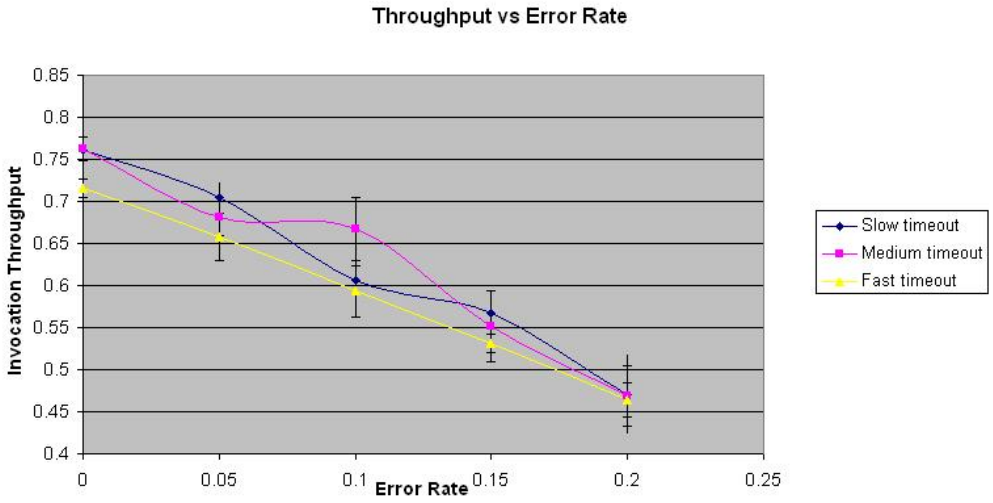

Figure 17: Throughput vs Error Rate for ESRO modelled by proSPEX

determine whether the ESRO server could deliver a required throughput and to examine the use of queuing performance metrics for system tuning. The server was shown to be able to deliver the required throughput.

In order to increase the load on the server we adapted the model and in the process drastically reduced the computational power of the server. The resultant queueing statistics, mean and maximum queue length and mean queue waiting time, revealed that relatively insignificant system resources, such as buffer space, is consumed by the ESRO server.

In our final experiment we compared throughput when the protocol is deployed in a network environment with five different loss probability settings. In this experiment we observed variations in the optimal service data unit retransmission setting as the error rate increased.

Chapter 8

Conclusion

8.1 Summary

With enhanced real-time architectural specification abilities and semantic tightening using profiles, UML 2.0 appears poised to become the dominant specification language used in real-time modelling and protocol engineering. In the field of protocol performance analysis, a fair amount of research has been conducted using mature protocol engineering languages, such as SDL, however this has not been the case with UML 2.0. Consequently, in this dissertation we have investigated protocol engineering and performance analysis using UML 2.0.

Our primary aim was to develop a prototype performance analysis tool which could translate from a UML 2.0 protocol specification to an equivalent simulation model by integrating existing tools. With our tool, proSPEX (protocol Software Performance Engineering using XMI), we have used approaches to performance analysis using both SDL and UML as a foundation. Performance analysis using both languages was applicable since we have used UML 2.0 enhanced with the ITU Z.109 profile "SDL Combined with UML".

With UML 2.0 introducing many more diagram types that are found in more established protocol engineering languages such as Estelle, SDL and PROMELA, we found the need to develop a proSPEX methodology. In this methodology we detailed procedures and language features used in protocol design, specification and the specification of non-functional, or *temporal*, delay constraints. Our methodology entailed using a combination of class, architecture and state chart diagrams for protocol specification. Environmental characteristics, such as network link bandwidth, delay distribution and loss probability, are specified

using a collaboration diagram in which a performance scenario is specified. The final step in our methodology was to integrate Telelogic Tau, our model editor, and Simmcast, a discrete event simualtion framework. Thus, the proSPEX tool generates quantitatively assessible simulation models from protocol models specified using Telelogic Tau and the proSPEX methodology by using the Tau XML model storage format for model interchange.

In the implementation of proSPEX, we found the need to extend the Simmcast simulation framework in order to be able to generate simulation Java code that represents a UML 2.0 protocol model as specified using the proSPEX methodology. In the proSPEX extension to Simmcast, which is effectively a SDL runtime support system, we developed means of representing finite state machines, SDL Pid expressions, SDL implicit addressing, the system architecture and additional trace events. The automated translation from a Tau model to an executable simulation model took part in two phases. In the first phase, data structures are populated by filtering the Tau XML. In the second phase, these data structures and a set of text templates are merged in a code generation process, using the Apache Velocity text templating engine.

We found that the development of the proSPEX filter was severely hampered by a lack of supporting documentation detailing the syntax and semantics of the Tau XML. This resulted in a manual process in which we uncovered the meaning of the Tau XML elements using intuition and trial and error. In order for the realization of the vision [MTMC99] of better and stable performance analysis tools, that interoperate with commercial SDL/UML 2.0 tools, well-documented interchange standards such as XMI 2.0 will have to be implemented by vendors.

In order to demonstrate the utility of proSPEX, we applied the proSPEX methodology and approach to performance evaluation to a wireless e-commerce scenario in which the Efficient Short Remote Operations (ESRO) protocol is used as a reliable transport protocol. The ESRO model was parameterized with time-out settings, a service data unit size, network link characteristics, connector buffer sizes and nodal processing delay parameters. These experiments showed that the ESRO server was able to deliver the required throughput. In addition, the experiments showed that with an estimated maximum queue length of 7 packets at the server and a maximum PDU length of 1500 bytes, the largest buffer size required in the ESRO server would be about 10kB.

In general, we believe proSPEX achieved the objectives initially set for it.

8.2 Future Work

A number future developments to the proSPEX tool, semantics time model and methodology that would be beneficial in future are listed and discussed below.

1. *The proSPEX methodology*: in the proSPEX methodology we have used UML class, architecture and collaboration diagrams. As we know, the primary purpose of these diagrams is to aid in understanding and unambiguous communication. In the context of the goals of this dissertation, UML 2.0 architecture diagrams have not offered significant advantages over SDL architecture diagrams. In future we would recommend the use of pure SDL (SDL/GR and SDL/PR), as described in the ITU Z.100 standard and not by a single tool vendor, to describe the architecture and dynamic aspects of the system and UML to describe static objects and their relationships. This approach is taken by Pragmadev [Pra04] in their recently released RTDS G3 tool.

2. *The proSPEX implementation*: in implementing proSPEX we found the need to extend Simmcast to allow for the representation of various high-level SDL finite state machine abstractions. In a future version of the tool the mapping to an efficiently implemented (C, C++) industrial simulation tool, with built-in finite state machine abstractions, should be considered.

3. *Performance analysis tool requirements analysis and industrial case-studies*: there is clearly a need for performance analysis tools that interoperate with industrial modelling tools [MTMC99]. In addition, industrial case studies are required in order to test important aspects such as whether usability and functionality requirements are met. Such aspects are not mentioned or considered in most related work in which SDL is used as the specification language.

Appendix A

An Introduction to UML 2.0

In this chapter we consider the diagrams and features of UML 2.0 that are of particular importance in the field of protocol engineering. We use the Telelogic Tau/Developer Generation 2.1 tool to visualize the described diagrams where possible.

In Section A.1 and Section A.2, we examine the UML 2.0 diagrams that are particularly well suited to protocol engineering. We then look at model-driven development in Section A.3 and lastly XMI 2.0 in Section A.4.

A.1 UML 2.0 Composite Structures

In this section we investigate the enhancements to the UML architectural modelling capabilities. The UML 2.0 composite structures subpackages represent major enhancements to UML in terms of its architectural specification abilities, which are particularly important[SR03b] in the real-time domain. The features are primarily derived[Sel03] from both ROOM[SGW94] (Real-Time Object-Oriented Modelling) and SDL. The composite structures subpackages of UML 2.0 along with the role of collaborations are listed below.

- **InternalStructure subpackage:** "...provides mechanisms for specifying structures of interconnected elements that are created within an instance of a containing classifier [class]."[Gro03]
- **Ports subpackage:** "...provides mechanisms for isolating a classifier [class] from its environment."[Gro03]

- **Collaborations:** A collaboration diagram offers a view of cooperating instances achieving a joint task, with each instance playing a particular role in the interaction. Connectors between participating instances specify the communication paths that must exist to enable collaboration.
- **Structured Classes subpackage:** "...supports the representation of classes that may have ports as well as internal structure."[Gro03]
- **Actions subpackage:** "...adds actions that are specific to the features introduced by composite structures, e.g., the sending of messages via ports."[Gro03]

We investigate the most important aspects[1] of the composite structure packages in this section, with a focus on active classes throughout.

A.1.1 Active and Passive Classes

The class is the fundamental classifier in the object-oriented paradigm and is specified in the *Kernel* package in the UML 2.0 Specification. The set of objects that are of a particular class can either be passive or active. In essence an object of an active class has its own thread of control and an object of a passive class is an information store (i.e. used for describing data structures[LTB98]). The comprehensive OMG definitions[Gro03] are as follows:

class: A classifier that describes a set of objects that share the same specifications of features, constraints, and semantics.

active object: An object that may execute its own behavior without requiring method invocation. This is sometimes referred to as "the object having its own thread of control." The points at which an active object responds to communications from other objects are determined solely by the behavior of the active object and not by the invoking object. This implies that an active object is both autonomous and interactive to some degree.

Notationally, an active class is distinguished from a passive class by vertical bars being present on the side of an active class as shown in Fig. 18.

[1] See Chapter 9, *Composite Structures*, of the UML 2.0 Superstructure Specification[Gro03] for comprehensive details.

Figure 18: Active class notation.

A.1.2 Provided and Required Interfaces

Interfaced-based design has the benefit of both reduced design complexity and giving distributed teams the ability to work concurrently while using the interface as a contract. In UML 2.0 an interface is a classifier representing a declaration of a set of public features and obligations[Gro03]. Interfaces are not instantiable, instead they are either *provided* or *required* by a classifier such as a class. When a class provides an interface it carries out its obligations to clients of instances of the class. When a class requires an interface it means that it needs the services specified in the interface in order to perform its function and fulfill its own obligations to its clients.

Protocol engineers experienced in the use of SDL would be accustomed to signal exchange between SDL processes via channels (when the processes are different blocks) or signalroutes (when processes are in the same block). In addition the direction of signals are clearly indicated in the SDL architecture diagrams. In UML 2.0 a required interface of an active class represents the signals it can expect from its environment whilst the provided interface represents signals it can send to its environment. The environment is either a container class or peer class, e.g. two peers communicating via a network link. The UML 2.0 architecture diagrams, which hierarchically decompose active classes, show the direction of signal exchange between the parts of the active class. Hierarchical decomposition is discussed in Sect. A.1.4.

As can be seen in Fig. 19, an architecture diagram, the notation introduced for a provided interface is a full-circle lollipop whilst the notation introduced for a required interface is a semi-circle lollipop. The squares to which the required and provided interfaces are connected are ports, which we examine next in Sect. A.1.3.

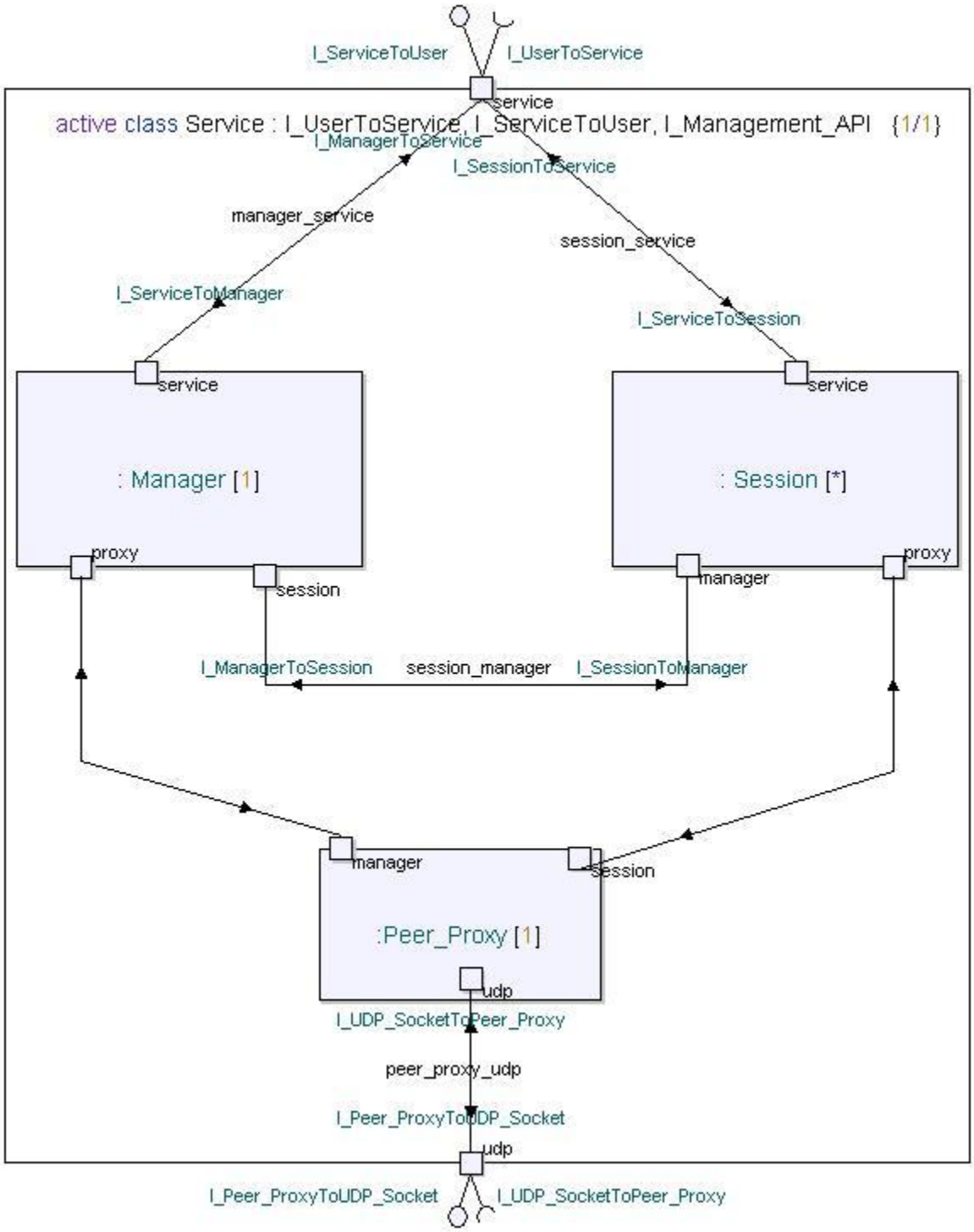

Figure 19: UML 2.0 Architecture Diagram.

A.1.3 Ports

Ports serve the purpose of being used to group an active class's related interfaces and also act as interaction (or connecting) points through which the services of a class can be accessed. In architectural diagrams different active classes are connected via ports serving as interaction points. A class with ports has its ports as its sole interaction points, meaning that any interaction with the class is message based and not via public operations or attributes.

It is important to note that ports have the role of decoupling an active class from its environment and therefore allow the active class to be defined independently of its environment. This independence makes the active class reusable in any environment that obeys the interaction constraints enforced by its ports[Gro03].

In Fig. 19 we see a bi-directional port (service) attached to the Service active class which has one required interface (I_UserToService) and one provided interface (I_ServiceToUser). A port may also be uni-directional and if it has a single provided interface this means that calls can be sent to the class and return values received. Calls can be sent from a class with a required interface to another providing the interface and return values received[LTB98].

A.1.4 Internal Structure with Parts and Connectors

In UML 2.0 particular attention has been given to hierarchical decomposition of classes, that is, the ability to specify the internal structure of classes in a modular fashion. Active classes can not only have ports and interfaces but also internal structure. An architecture diagram (called *composite structure diagram* in UML 2.0) representing the internal structure of an active class can be defined as *an abstract metaclass whose behaviour can be fully or partly described by a collaboration of owned or referenced instances*[Gro03].

The graphic nodes included in architecture diagrams are parts, ports, collaborations and collaboration instances. An example of an architecture diagram of an active class is given in Fig.19. Note that the *ports* of the containing instance's *parts* (the collaborating instances) are linked with *connectors*. A connector specifies a communication link between two or more instances. It is important to note that the link could represent an instance of an association (as specified in class diagrams) or the mere possibility of the instances being able to communicate. This possibility may be borne out of the linked instances knowing each other's identities due to being passed in as parameters, held in variables, created dynamically, or since the communicating instances are the same instance. In the concrete

software which the model represents the link could be realized as a reference or a network connection. The difference between a connector and association is that a connector specifies links that exist solely between instances playing the connected parts whereas associations specify links that exist between any instances of the associated classes[Gro03].

In Fig.19 also note the multiplicities of the parts. Multiplicities specify the number of instances, both objects and links, that may be created within an instance of the containing classifier. The instances are either created when the containing class is created or at a later time.

A.1.5 Behaviour Ports

In UML 2.0 we have a black box view and a white box view of a class. With the black box view we only see the provided and required interfaces. With the white box view the class's implementation is revealed, that is, we see the decomposition of the containing class into its *parts* and the *connectors* joining them, as we saw in Sect. A.1.4. In the black-box view we want to be able to distinguish between behaviour that is delegated to the class itself and behaviour that is delegated to its parts. Connectors terminating in a behaviour port mean that the signals sent to the port are handled by the containing class. Notationally a behaviour port is represented by a state symbol attached to a square port symbol. Behaviour ports are particularly common when a state machine represents the behaviour[BK03]. Port Manager_Internal in Figure 20 is a behaviour port.

A.2 UML 2.0 Behaviour Descriptions

A.2.1 Overview

In this section we provide a brief overview of the behavioral diagrams most commonly used in communication protocol engineering, namely sequence and state machine diagrams. We also mention the protocol specification abilities that have been introduced in UML 2.0. It should be noted that most of the features mentioned in this section were not implemented by any UML tool vendor during the duration of this dissertation[2].

[2]The UML 2.0 tool available during the duration of this thesis was Tau Generation 2.1 which implements a proprietary UML profile (Tau-RT), as discussed in Section 3.4

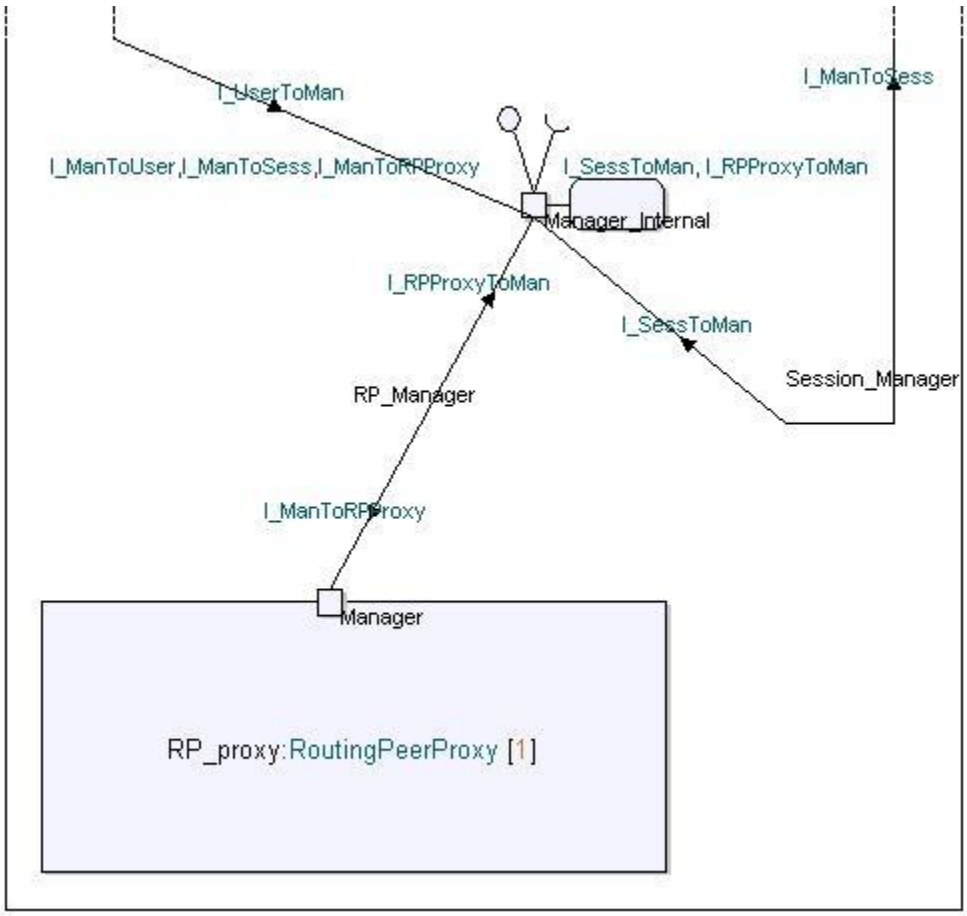

Figure 20: UML 2.0 behaviour port notation.

Interactions

Sequence Diagrams fall under a set of UML 2.0 diagrams called *interactions*. The Sequence Diagram, Communication Diagram, Interaction Overview Diagram, Timing Diagram and Interaction Table are all *interactions*. In the detailed design phase interactions may be used to specify inter-process communication, but this is often for understanding purposes. In the testing phase traces (event occurrence sequences) of system execution can be visualized as interactions.

Interactions focus on the (asynchronous or synchronous) communications between instances in a collaboration communicating using messages. All interactions occur in the structural context of the collaborating parts[Sel03]. The reader is encourage to consult the UML 2.0 Superstructure Specification[Gro03] for further details of interaction diagrams.

State Machine Diagrams

The most significant enhancements to finite state machines (FSM) in terms of new modeling constructs that have been introduced are listed below.

- **Modularized submachines**
- **State Machine Specialization/redefinition**
- **State Machine Termination**
- **Protocol State Machines**

The most significant enhancements to notational elements are listed below.

- **Action blocks**
- **State Lists**

During this disseration, none of the mentioned UML 2.0 FSM constructs have were implemented by tool vendors. Telelogic Tau, the model editor used in this dissertation, has integrated SDL state machines into UML 2.0.

A.2.2 Actions

Model-driven development with executable models is at the heart of the Model Driven Architecture (MDA) initiative of the OMG. In order to make models executable, actions must be specified at a level of granularity comparable to most programming languages[LTB98]. As mentioned in Section 3.4 UML is not *complete*, meaning that it cannot be used directly as a programming language. In order to use it as a programming language *profiles* are needed to tighten the semantic variation points (from which the language derives its flexibility) along with a data model with basic data types. The profile should include an action syntax in order for users to be able to use the actions included in the profile[LTB98]. The Tau-RT profile, discussed in Section 3.4 has all of the features mentioned above that turn UML into a programming language.

The advantages of executable models are:

- **Model verification**: The correctness of the system can be at the early stages of development before code is produced.
- **Verification and validation technique application**: With executable models techniques such as state space exploration can be used to verify certain functional properties of the system such as being free from deadlock or livelock.
- **Automatic code generation:** With the executable model being at a higher level[3] than most programming languages it is possible to generate code for a variety of platforms from the model.
- **Performance Analysis:** Execution traces can be used to derive performance statistics used to detect performance problems at an early stage of development.

A.3 Model-Driven Development

Although it is desirable for models to be executable one should note that model-driven development does not imply that the models are executable. Definitions of model-driven development make it clear that the essential ingredient is having a model as the focus of the development effort, as the following two quotes show. "The term model-driven development implies that a model is at the centre of development, and is used as the basis

[3] A model is implicitly an abstraction of the real system and therefore at a variably higher level.

for application development."[LTB98]. "...the primary driving force behind UML 2.0 is model-driven development, an approach to developing software that shifts the focus of the development from code to models while automatically maintaining the relationship between the two"[Sel04].

A.4 XML Metadata Interchange Format 2.0

The XML Metadata Interchange (XMI) Format was originally developed as a standard vendor-independent way for UML tools to interchange UML models. Certain UML modelling tools, such as Poseidon, save UML models using XMI as their native format while others, such a Rational Rose, have the ability to save and load XMI models as a foreign format. Unfortunately incompatibilities between XMI written by different vendors exist and is partly due to shortcomings[Ste03] in the standard itself. With XMI 2.0, the diagram interchange format of UML 2.0, the OMG has addressed the short comings of the earlier XMI standard.

Appendix B

Patterns for Protocol System Architecture

The common parts found in communication protocol system design and resultant implementation can be described using patterns. Patterns are associated with object-oriented design and hence can be readily applied and described using SDL or UML. Generally patterns are described using UML as is done in this section.

Patterns provide *common principles* for not only designing and implementing new protocols but also for the general understanding of protocols and their parts. *Patterns*, *mechanisms* and *frameworks* operate hand-in-hand, as explained by Booch, Jacobson and Rumbaugh[BRJ98]:

> *A **pattern** provides a common solution to a common problem in a given context. A **mechanism** is a design pattern that applies to a society of classes. A **framework** is typically an architectural pattern that provides an extensible template within a domain.*

We use patterns to specify the frameworks that shape communication protocol architecture. These architectural patterns, or frameworks, are manifested at varying levels of abstraction and so a stereotyped package representing the framework may include classes, interfaces, use cases, components, nodes, collaborations and other frameworks[BRJ98]. In this section we provide a brief overview of three closely related patterns for protocol system architecture created by J. Parssinen and M. Turunen and detailed in [PT00][1].

[1] We borrow from [PT00] and the interested reader is urged to consult this more comprehensive paper.

The patterns we describe are the Protocol System pattern, the Protocol Entity pattern and the Protocol Behaviour pattern. The Protocol System pattern models the protocol system at a general, high level, while the Protocol Entity pattern models the active system components. Lastly the Protocol Behaviour pattern models communication between the protocol system components.

The *benefits*[PT00] of an understanding and application of the three patterns mentioned in this section are:

- An architecture which is clear and hence readily understandable, consistent and efficient to both implement and maintain.
- The effort-less integration of protocol components produced using different implementation frameworks.
- Aid in developing protocol implementation tools.

B.1 Communication Protocol Structure

A reference diagram[PT00] which encapsulates the common elements found in protocol implementations is shown in Figure 21. The diagram has similarities to the ISO Seven Layer Reference Model in which each layer in the protocol stack solves a different problem or set of problems. Since the best frameworks are derived from existing architectures which are proven to work, the similarity to the Reference Model is not surprising. The reference diagram established the vocabulary used in the patterns described in this section.

B.2 Protocol System Pattern

The Protocol System and Entity patterns model the static parts of a protocol system, that is the components and their interconnections. The Protocol Systems pattern gives the highest level view of the components of a protocol system as well as the interfaces to its environment.

As one can seen in Figure 22, a *Protocol System* is composed of one or more *Protocol Entities* each of which represents a protocol layer or sublayer. The *Entity Interfaces* define the allowed incoming and outgoing messages exchanged between Protocol Entities within the same system. A Protocol System has one or more *Environment Interfaces* and these act as a

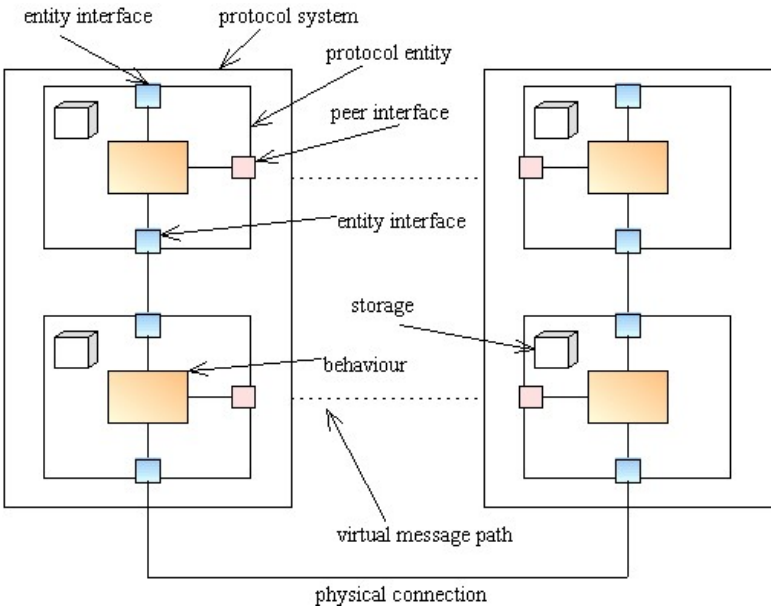

Figure 21: Protocol implementation elements

message source for incoming external messages and as a message sink for outgoing messages. An Environment Interface has Communication Interface and Auxiliary Interface subclasses. A Communication Interface specifies the messages related to normal communication with the environment while an Auxiliary Interface specifies test and management messages.

B.3 Protocol Entity Pattern

A *Protocol Entity* can send and receive messages and has an inner state while doing so. Each entity has a *virtual* message path to its peer entity and real message paths to its adjacent entity or environment as shown in Figure 21.

The Protocol Entity pattern, shown in Figure 23, contains the *Protocol Entity*, *Storage* and *Protocol Behaviour*. Each Protocol Entity requires (uses) and provides *Entity Interfaces* and *Peer Interfaces*.

The *Protocol Entity* is a single protocol layer and the *Protocol Behaviour*, detailed in the Protocol Behaviour Pattern in Section B.4, is the source of protocol functionality. The *Storage* contains all the persistent and non-persistent data of a protocol entity. This data

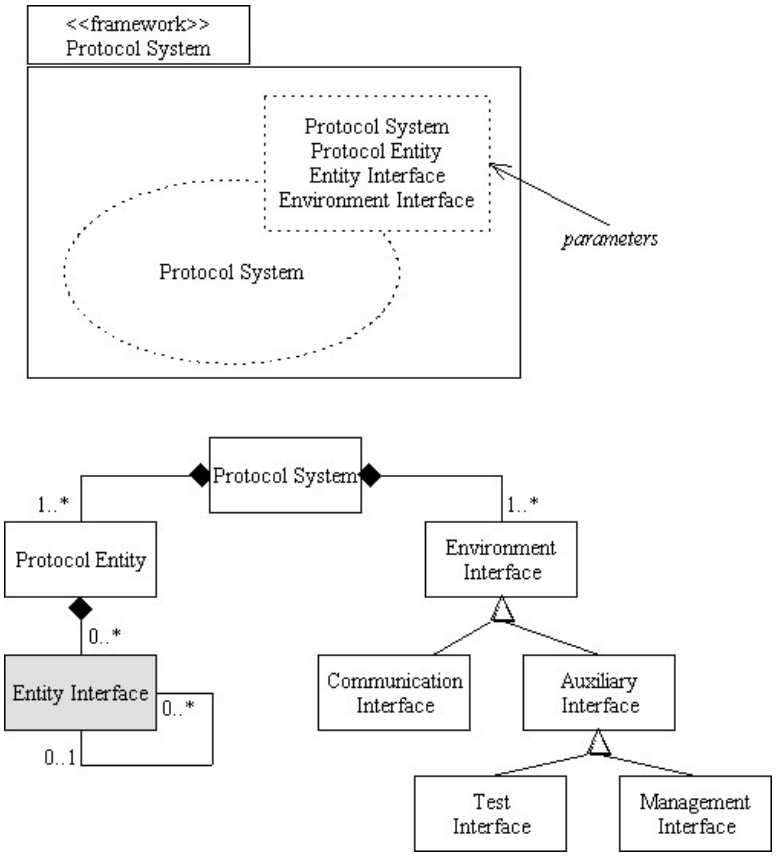

Figure 22: Protocol System Pattern

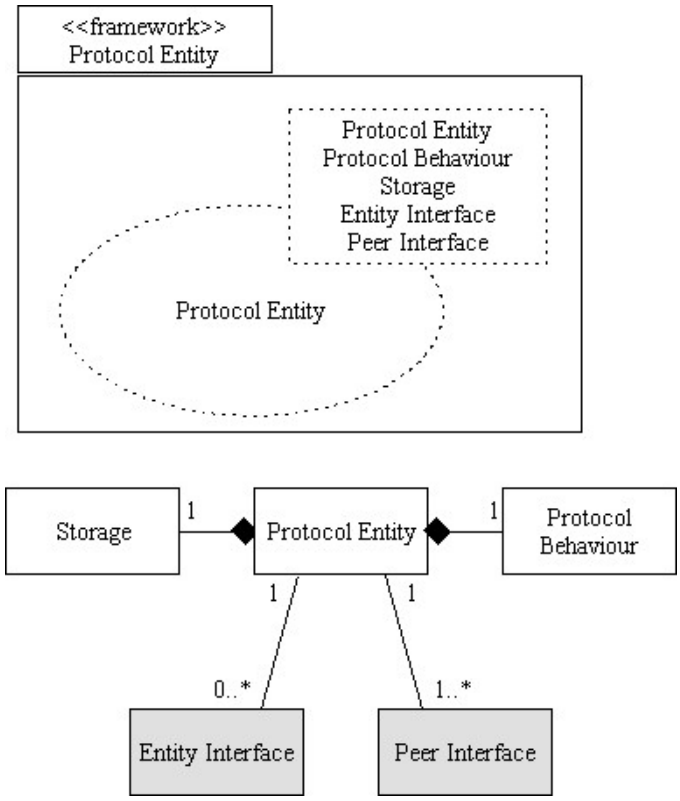

Figure 23: Protocol Entity Pattern

could be divided into communication session specific parts. Two Entities in the same protocol system communicate via an *Entity Interface*. The Entity implementing the Entity Interface interprets Entity messages received from another Protocol Entity and produces Entity messages sent to another Entity in the same system. Similarly a *Peer Interface* specifies the messages that are interpreted when received and sent by Entities in the same system.

B.4 Protocol Behaviour Pattern

The Protocol Behaviour Pattern can be used to implement entity functionality in a communication protocol system. The communication is either connectionless or connection-oriented.

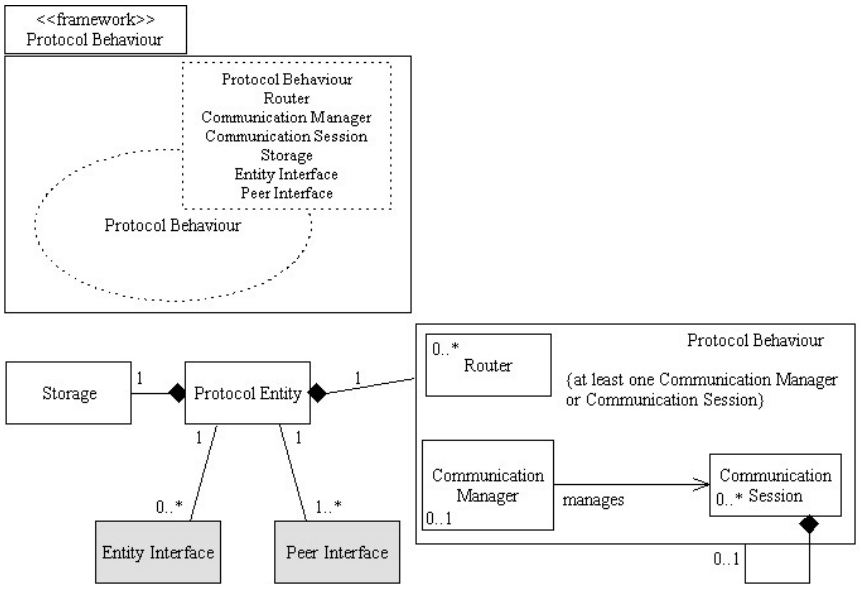

Figure 24: Protocol Behaviour Pattern

Figure 24 shows the Protocol Behaviour components, that is zero or more *Routers*, zero or one *Communication Manager* and zero or more *Communication Sessions*.

A *Router* is used when multiple *Communication Sessions* can receive messages coming from a single entity interface. The *Router* routes messages to the correct *Communication Session* or to the *Communication Manager*. A *Communication Manager* would mostly, but not exclusively, be used in connection-oriented protocols in which it would create, control and terminate Sessions as is required. Communication between peer entities is handled by *Communication Sessions*.

There is a distinction between the use of the Protocol Behaviour pattern in the case of connectionless and connection-oriented protocols. With *connectionless protocols* the Protocol Behaviour (see Figure 24) generally contains one *Communication Session* handling all communication. The exception to this rule is where the Protocol Entity should be able to serve multiple simultaneous requests in which each request could take moderately long. In this case the Protocol Behaviour has a *Communication Manager* which creates a *Communication Sessions* to serve each individual request.

With *connection-oriented* protocols the Protocol Behaviour generally contains two *Routers*, one *Communication Manager* and zero or more *Communication Sessions*. One *Router* is

used to route Service Data Units (SDUs) to the correct Protocol Entity user while the other routes Protocol Data Units (PDUs) to the correct *Communication Session*.

Appendix C

The Efficient Short Remote Operations Protocol

C.1 Introduction

The Efficient Short Remote Operations Protocol (ESRO) service, specified in RFC 2188 [Ned97], is similar to that provided by Remote Procedure Call (RPC) services. With the ESRO service and protocol the target network environment is wireless and hence the emphasis is on efficiency. ESRO provides a reliable connectionless remote operation service which sits on top of any datagram (non-reliable and connectionless) transport service, such as UDP. The ESRO service can be used with either a 2-way or 3-way handshake protocol, thereby offering varying degrees of efficiency and reliability.

C.2 The ESRO Service Definition

A user of ESRO either assumes to role of operation *invoker* or operation *performer* (see Figure 25) which in turn determines the behaviour and services offered by the two peer ESRO sublayers. Having received notice of an operation that is to be performed, the performer user is expected to report the result of the operation or an error. The operation at the invoker user would either have been successful or have resulted in an error and as a consequence a result or error reply would be delivered to the performer user. Operations are generally performed asynchronously although synchronous operations are also supported.

The ESRO service primitives are accessed through an ESRO Service Access Point

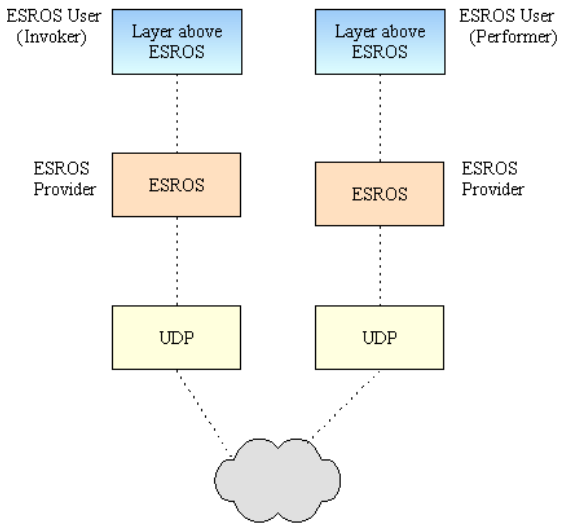

Figure 25: The ESRO Operation Model.

(ESRO-SAP). These service primitives are the ESROS-INVOKE.request, ESROS-INVOKE-P.confirm, ESROS-RESULT.request, ESROS-ERROR.request and ESROS-FAILURE.indication primitives. The sequence diagram in Figure 26[1] gives an overview of the use of the ESRO services.

In the event of no errors occurring at the ESRO provider or performer user, the usage of ESRO would be as follows. Firstly the ESRO Invoker user would, via its ESRO Invoker SAP, use the ESRO-INVOKE.request service to *request* a method invocation. The ESRO Provider would acknowledge receipt of the ESRO-INVOKE.request operation by delivering a Provider initiated ESRO-INVOKE-P.confirm message[2]. Once the ESRO protocol has successfully delivered the data using one or more UDP Packets, the ESRO Provider would deliver a method invoke indication message using the ESROS-INVOKE.indication primitive. Once the operation has been successfully executed, the ESRO Performer user would, via its ESRO Performer SAP, use the ESROS-RESULT.request primitive to *request* delivery of the result to the Invoker user. Having received the ESROS-RESULT.request, the ESRO Performer Provider would deliver the result to the ESRO Invoker Provider. The

[1]The sequence diagram in Figure 26 was generated by the popular Poseidon Tool by Gentleware AG. We have used the Poseidon Community Edition, which has the benefits of being freely distributed, storing diagrams exclusively using XMI 2.0 and saving diagrams in Encapsulated Postscript format.

[2]The *P* in ESRO-INVOKE-P.confirm stands for *Provider initiated*.

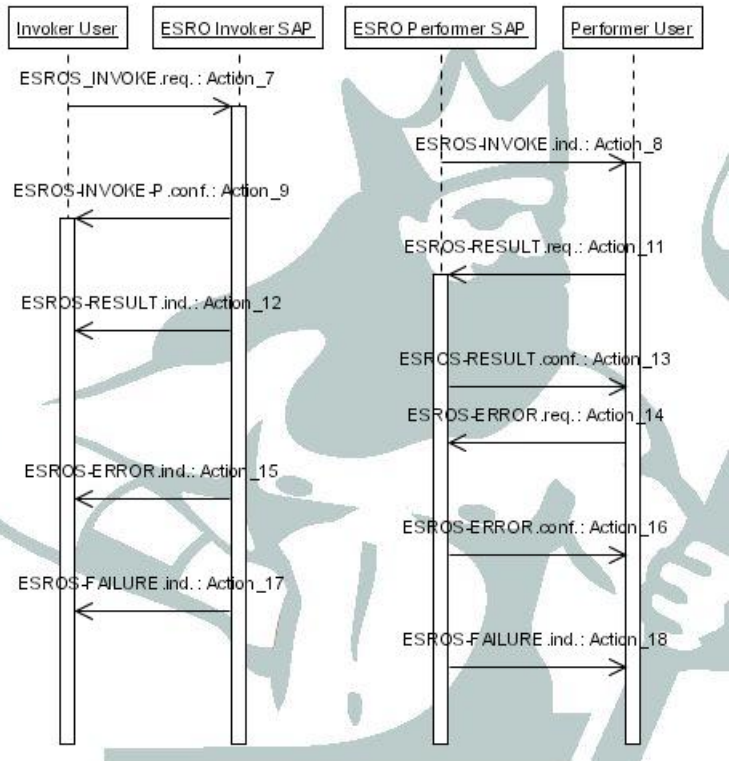

Figure 26: Sequence Diagram for ESRO Services.

ESRO Invoker Provider would then send an ESROS-RESULT-indication message to the Invoker User. Finally, the ESRO Performer Provider would send an ESROS-RESULT.confirm message to the Performer User.

In the event of error (which would occur either at the ESRO Performer User, the ESRO Provider, or due to network related eventualities) the ESROS-ERROR and ESROS-FAILURE primitives would be used.

C.3 The ESRO Remote Operations Protocol

The ESRO Services, which we mentioned in the previous section, are realized by the ESROS protocol. In order to use ESRO Services, the user binds to an ESROS SAP and specifies either a 2-Way or 3-Way handshake Functional Unit.

The protocol used for transmitting Service Data Units (SDUs) is extremely simple due

to its retransmission strategy. When an SDU is segmented into multiple PDUs, the retransmission strategy is not applied to individual segments. Instead, the loss of a single segment results in the retransmission of the entire SDU. The number of SDU retransmissions and the optimal retransmission interval is network dependant and should be based on network statistics. Further details of the ESRO protocol PDUs and finite state machines are provided in the ESRO RFC [Ned97].

Appendix D

The proSPEX Templates

D.1 The Main Template

The template `main.vm` is used to generate the simulation main file.

```
package $packageName;

import arjuna.JavaSim.Distributions.*;
import prospex.*;
import java.util.Vector;
import simmcast.trace.SimmcastTraceGenerator;
import simmcast.stream.FixedStream;
import simmcast.node.NodeVector;

public class $mainClassName {

    public static final int UNLIMITED = -1;

    /**
     * NETWORK and SOFTWARE PATH characterisitics.
     **/

    static int soft_pathCapacity = UNLIMITED;
```

```
static double soft_bandwidth = UNLIMITED;
static double soft_lossRate = 0.0;
static FixedStream soft_stream = new FixedStream(0.0);

static int net_pathCapacity = UNLIMITED;
static int net_bandwidth = $bandwidth;

#if( $distribLowB && $distribUppB )
static double net_delay_lower_bound = $distribLowB ;
static double net_delay_upper_bound = $distribUppB;
static UniformStream net_stream = new UniformStream(net_delay_lower_bound,
                                                    net_delay_upper_bound);
#else
static ${delayDistrib}Stream net_stream = new ${delayDistrib}
                                                    Stream($distribMean);
static double net_lossRate = $lossProbab;
#end

public $mainClassName() {

}

public static void main(String args[]) {

    try {

    /**
     * PRE-LIM STEP 1: Create the ProNode classes that are able to call
     *   the method network.addSoftwareNode
     ***/

    /**
     * STEP 1: Create the ProNetwork.
```

```
 ***/

ProNetwork network = new ProNetwork("$packageName");

/**
 * STEP 1.5 : Create simulation collaboration scenario (replicate what
 *                  simulation description file did).
 **/

SimmcastTraceGenerator tracer = new SimmcastTraceGenerator();
network.setTracer(tracer);

tracer.setFile("$traceResultFile");

network.nodes = new NodeVector();

/**
 * NOTES: The constructor of extended Node say Manager creates the
 * NodeThread.initializeNode() schedules the EventScheduler associated
 * with this node NOW.
 **/

// Since 2 detected in Collaboration diag, we have 2 copies
// of each.....

#foreach ( $nodeTuple in $nodeTuples )
$nodeTuple.NodeClassName $nodeTuple.NodeName = new ${nodeTuple.
                                                    NodeClassName}();
network.initializeNode($nodeTuple.NodeName,"${nodeTuple.NodeName}");
${nodeTuple.NodeName}.setNetworkId(network.obtainUnicastAddress());
${nodeTuple.NodeName}.setProNetwork(network);
network.nodes.addNode(${nodeTuple.NodeName});
network.tracer.node(${nodeTuple.NodeName});
```

```
#end

/**
 * Add SOFTWARE paths.......
 **/

#foreach ( $pathTuple in $softPathTuples )
${pathTuple.SourceNodeName}.addBidirectionalPath(${pathTuple.TargetNodeName},
                soft_pathCapacity,soft_bandwidth,soft_stream,soft_lossRate);
#end

/******/

 /**
  * Add NETWORK paths........
  **/

#foreach ( $pathTuple in $netPathTuples )
${pathTuple.SourceNodeName}.addBidirectionalPath(${pathTuple.TargetNodeName},
                    net_pathCapacity,net_bandwidth,net_stream,net_lossRate);
#end

/**
 * STEP 2: Add node daemon info to the created ProNetwork.
 **/

#foreach ( $netArchNode in $netArchNodes )
network.addNetworkArchNode("${netArchNode.ClassName}",${
                                                    netArchNode.DaemonBoolean});
#end
```

```
/**
* STEP 3: Add paths to the created ProNetwork.
***/
#foreach ( $netArchLink in $netArchLinks )
network.addNetworkArchLink("${packageName}.${netArchLink.FromClassName}",
                           "${packageName}.${netArchLink.ToClassName}");
#end

/**
* STEP 3: Add the signals and their associated target classes to
* the network's SignalTargetHashTable. This information is drawn
* directly from the UML 2.0 required interfaces. Note that if a class
* can receive 2+ signals with the same name, we append the type of the
* signal payload to the signal name (underscore too!).
***/

#foreach ( $sigTargTuple in $netSigTargTuples  )
network.sigTargetHashT.addSignalTarget("${sigTargTuple.SignalName}","
                    ${packageName}.${sigTargTuple.TargetClassName}");
#end

/**
* STEP 5: Run the simulation!
***/

network.runSimulation("network");

System.out.println("Done!");

} catch (Exception e) {

    System.err.println("Caught Exception: " + e.getMessage() +
                                              e.toString());
```

```
            e.printStackTrace();

        }

    }
}
```

D.2 The Node Template

The template `node.vm` is used to generate the Node classes. Each UML 2.0 active class is mapped to a Node and associated NodeThread class. The NodeThread class contains the protocol logic and is generated by the state machine template.

```
/*
 * @(#)${className}.java
 *
 */
package $packageName;

import prospex.ProNode;

public class $className extends ProNode{

    /**
    * Thread that runs the server logic.
    */
   ${className}Thread thr;

    public ${className}() {
        thr = new ${className}Thread(this);
    }

    public void begin() {
      thr.launch();
```

```
    }
}
```

D.3 The Signal Parameter Class Template

The template `signalparam.vm` is used to generate classes that form the payload of signals.

```
/*
 * @(#)${signalPayload.ClassName}.java
 *
 */

package $packageName;

class ${signalPayload.ClassName} #if ( $signalPayload.Parent != ""
) extends $signalPayload.Parent #end {

    #foreach ( $attrib in $signalPayload.Attribs )
    public $attrib.AttribType $attrib.Name;
    #end

    #if ($signalPayload.Networked)
    public int length;
    #end

    public ${signalPayload.ClassName}() {

    #foreach ( $attrib in $signalPayload.Attribs )
    #if ( $attrib.DefaultValue != "" )
    $attrib.Name = $attrib.DefaultValue;
    #end
    #end

    }
```

```
    /*****************************
    *
    * GETTERS / SETTERS
    *
    *****************************/

    #foreach ( $attrib in $signalPayload.Attribs )
    public $attrib.AttribType get${attrib.Name}() {
        return $attrib.Name;
    }

    public void set${attrib.Name}($attrib.AttribType ${attrib.Name}_) {
        $attrib.Name = ${attrib.Name}_;
    }
    #end

    #if ($signalPayload.Networked)
    public int getlength() {
        return length;
    }
    public void setlength(int length_) {
        length = length_;
    }
    #end


}
```

D.4 The State Machine Template

The template `statemachine.vm` is used to generate state machines.

```
/*
```

```
 * @(#)${className}Thread.java
 *
 */
package esro_invoke;

import simmcast.node.*; import simmcast.network.*; import
java.util.*; import java.text.*; import prospex.*;

// NB: Purely due to use of addSoftwareNode
import java.lang.reflect.*;

/**
 * Included to solve division prob.
 **/
import java.lang.Math;


class ${className}Thread extends NodeThread {

    ${className} myself;

   /*******************************************
    *
    * STATE MACHINE CONTROL VARIABLES
    *
    *******************************************/

    private int state = $initialStateNumb;
    private int previous_state = 0;
    private int event = 0;

    #if ( $currentSMDat.InitialTask )
    private static final int STATE_0 = 0;
```

```
#end
#foreach ( $state in $states )
private static final int STATE_${state.Name} = ${state.StrNumber};
#end

#foreach ( $signal in $currentSMDat.Events )
private static final int EVENT${signal.EventNumber}_${signal.Name}
                                          = ${signal.EventNumber};
#end

$currentSMDat.Daemon

#if ( $currentSMDat.Daemon == "false" )
/* The variable done is used in non-daemon active classes.
 * It tells the state machine
 * when to stop waiting for input.
 **/
boolean done = false;
#end

/* Need a generic time-out to be used in onTimer.
 **/
PacketType genericTimeOutPacketType = new PacketType("genericTimeOut");

/*****************************************************
*
* EXPLICIT STATE MACHINE VARIABLES (FROM UML 2.0 SPEC)
*
*****************************************************/

/* IMPORTANT: Code generator correlates variable names (below) and variable
 * types in order
 * to generate the.....
```

```
     * 1. event variable names
     * 2. PacketTypes objects
     * 3. parameters for this.myself.getSignalNetIdTarget("PeerDatagram_Invoke_PDU")
     *
     * proSPEX rule: variables that are classes are instantiated here, when declared.
     **/
#* NB: NO timers.... *#

#foreach ( $attrib in $currentSMDat.Variables )
 #if ($attrib.IsTimer == false)
  #if ($attrib.IsArray)
   $attrib.getAttribType() [] $arrib.getName() = new $attrib.getAttribType()
   [${attrib.DefaultValue}];
  #elseif ($attrib.DeclaredNew)
  #if ($attrib.DefaultValue) ## true if not null
   $attrib.getAttribType() $attrib.getName() = new $attrib.getAttribType()
   (${attrib.DefaultValue});
  #else
   $attrib.getAttribType() $attrib.getName() = new $attrib.getAttribType() ();
  #end
#else
 #if ($attrib.DefaultValue) ## true if not null
    $attrib.getAttribType() $attrib.getName() = ${attrib.DefaultValue};
 #else
    $attrib.getAttribType() $attrib.getName();
 #end
 #end
#end
#end

    /*******************************************************
     *
     * IMPLICIT STATE MACHINE VARIABLES (FROM SDL)
```

```
*
*****************************************************/

int source = 0; // our own address used when creating packets.

int sender = 0; // the address of the node/process that sent the last signal.

int offspring = 0; // the address of the latest child node that this class
                   // created.


/*****************************************************
 *
 * IMPLICIT STATE MACHINE VARIABLES (FROM DIAGRAM)
 *
 *****************************************************/

 /* NOTE: The names below are derived from the interfaces.
  * If there are two signals with the same name we simply
  * append the type of the signal payload to the name to
  * make it unique.
  **/

 /* RECEIVING:
  **/
 #foreach ( $signal in $currentSMDat.getEvents() )
 PacketType $signal.getName() = new PacketType("${signal.getName()}");
 #end

 /* SENDING:
  **/
 #foreach ( $signal in $currentSMDat.getSendSignalsHashT() )
 PacketType $signal.getName() = new PacketType("${signal.getName()}");
```

```
   #end

 /*********************************************
  *
  * proSPEX variables
  *
  *********************************************/

  /* NOTE: "length" is a field that is cumpolsory for classes that form
  * the payload of packets traversing network links, it is used when creating
  * the Simmcast Packets before they are sent.
  * By implication proSPEX must keep track of whether
  * a signal travels across network links or not!!! If a Packet does
  * not cross network links its length is set to SOFTWARE_PACKET_LEN
  **/

  private static final int SOFTWARE_PACKET_LEN = 0;


 /*********************************************
  * VARIABLE END
  *********************************************/


public ${className}Thread(Node node_) {
   super(node_);
   myself = (${className}) node_;
   daemon = ${daemon};
}

/*************************************************/

 public void execute() throws TerminationException {
```

```
setName("${className}Thread " + node.getName());
DecimalFormat f2d = new DecimalFormat("0000");
System.out.println("Executing ${className} thread " +
node.getName() + ", node " + node.getNetworkId());

source = myself.getNetworkId(); // our own address used when creating packets

/*******************************************
 *
 * START STATE MACHINE CONTROL
 *
 *******************************************/

  while (done == false) {

          switch (state) {

          #if ( $currentSMDat.InitialTask )
          case STATE_0: {

          /* A call to initialStateActions can only appear in the transition
          * to the initial state. Such a call only occurs if actions are
          * taken BEFORE we enter the initial state. It is then possible
          * to wait for signals/Events in the initial state.
          **/
                  previous_state = state;
                  state = initialStateActions();

              }
              break;
          #end
          #foreach ( $state in $currentSMDat.getStates() )
```

```
case STATE_${state.getName()}: {

        // determine what the next input signal/packet is....
        Event event = getInputSignal();

        switch(event.getType()) {

        #foreach ( $signal in $state.getAssociatedEventSignals() )

        case EVENT${signal.getEventNumber()}_${signal.getName()}:

        previous_state = state;

        state = processEvent${signal.
        getStrEventProcessingNumber()}_${signal.getName()}(event);
        break;
        #end

        default:
        this.myself.network.tracer.signal_discard(
        event.getPacket(),this.state);
        }
    }
    break;
#end

default: {
        throw new StateMachineException("There is
                                    no default state.");
    }
} // switch

/* Here we have a successful transition from one state to another.
```

```
                 * We record this in the trace.
                 **/
                this.myself.network.tracer.state_transition(this.myself,
                                                        previous_state,state);

        } // while(true)

        System.out.println("Session is DONE!");


       /***********************************************
        *
        * END STATE MACHINE CONTROL
        *
        ***********************************************/

    }

/**
* Blocks waiting for a signal, once it receives something it checks what
* type of signal it is, the type of the signal determines what EVENT is
* returned. It is possible for a state machine to receive a signal in a
* state that it cannot process in that state, such signals are discarded
* (ala SDL semantics). These semantics are implemented in the state machine
* control in function "execute" by doing nothing by default (when unexpected
* signal arrives) in each state.
**/

Event getInputSignal() throws TerminationException, StateMachineException {

    Packet packet = receive();

    // Set the SDL variable....
```

```
sender = packet.getSource();

System.out.println(packet.getType().toString());

/* The packet that we receive here could be a time-out packet. In which case
* we would have a generic time-out packet. The object contents of the packet
* would tell us exactly what type of time-out occurred.
**/

/**
* NOTE: The Packet_Type(s) used when sending packets in all active class
* threads must be correlated with the text in this getInputSignal
* function.
**/

#set($counter = 0)
#foreach ($signal in $currentSMDat.Events )
#if ($signal.getIsTimer())
#if ($counter == 0)
if (packet.getType().equals((Object)genericTimeOutPacketType)) {

/*
* We know that some time-out occurred. But we need to create an event
* letting the state machine control know exactly the type of the time-out
* occurred. This information is contained as object payload (of type String)
* of the received packet. This String must match with one of the PacketType
* variable of this class.
**/

String theRealType = (String)packet.getData();

Event returnMe = null;
```

```
/*
* TODO: Fill in the time-out options below!
**/
if (theRealType.compareTo("${signal.getName()}") == 0) {

/* Passing the packet argument as null, since it has served its purpose.
**/
returnMe = new Event(null, EVENT${signal.getEventNumber()}_${
                                                    signal.getName()});

}
#else
else if (theRealType.compareTo("${signal.getName()}") == 0) {

returnMe = new Event(null,EVENT${signal.getEventNumber()}_${
                                                    signal.getName()});

}
#end
#set($counter = $counter + 1)
#end
#end
#if($counter > 0)
else {

}
return returnMe;

}
#end

#foreach ($signall in $currentSMDat.Events )
    #if ($signall.getIsTimer() == false)
```

```
    #if ($counter == 0)
        if (packet.getType().equals((Object)${signall.getName()})) {

           Event returnMe = new Event(packet,EVENT${signall.
                           getEventNumber()}_${signall.getName()});
            return returnMe;

       }
       #set($counter = $counter + 1)

        #else
            else if (packet.getType().equals((Object)${
                                          signall.getName()})) {

                Event returnMe = new Event(packet,EVENT${signall.
                           getEventNumber()}_${signall.getName()});
                return returnMe;

            }
        #end
    #end

#end
#if($counter != 0)
else {

    this.myself.network.tracer.signal_discard(packet,this.state);

    /**
     * We received a bogus signal and reported it. Now we must
     * return a valid Event, so we call the method we are in again
     **/
```

```
        return getInputSignal();
     }
    #end
}

public void onTimer(Object message_) {

  this.myself.network.tracer.time_out(myself,this.state,
                                                  message_.toString());

  /*
   * When onTimer is called, this thread must be blocking waiting
   * to receive a packet. We want to stop it from blocking by magically
   * inserting a TIMEOUT packet with a particular name into this node's
   * RQ......see EventScheduler.manageEvent
   */


   /* 1. ADD Packet to Node's receiverQueue.
    **/

     // send a time-out Packet to myself...

    int destination = source;

    /* The message_ contains a String telling exactly which type of time-out
     * occurred.
     **/
    Packet timeOut = new Packet(source, destination,
                                       genericTimeOutPacketType,0, message_);

    // the idea for using the code below comes from EventScheduler.manageEvent
    // where its dealing with an ArrivalEventItem
```

```
    myself.receiverQueue.enqueue(timeOut);

    /* 2. UNBLOCK waiting NodeThreads.
    **/

    Iterator iter = myself.scheduler.waitingList.iterator();

    while (iter.hasNext())
        ((NodeThread)iter.next()).unblock();

    /* 3. Clear the waiting list.
     **/
    myself.scheduler.waitingList.clear();

}

/*********************************************
 *
 * END STATE MACHINE CONTROL Functions
 *
 *********************************************/


/*****************************************************************
 *
 * EVENT Processing Functions
 *
 *****************************************************************/

 #foreach ($state in $currentSMDat.getStates())

    #foreach ($transition in $state.getTransitions())
```

```
#set($startState = ${transition.getStartState()})
#set($eventSig = ${transition.getEventSig()})

/* Processed in State: STATE_${startState.getName()}
* Next state: ?
**/
public int processEvent${eventSig.getStrEventProcessingNumber()}_${
    eventSig.getName()}(Event event_) throws NodeNotFoundException,
                                             proSPEXUsageException,
                                             TerminationException,
                                             ProNetworkSetupException  {

    /* Must set the state at end of function...
    **/
    int nextState = -1;

#* Need to put parameters in place...first get the type of the signal,
* ie is it a timer?
*#
#foreach ($param in ${eventSig.getEventParameters()} )
    We have param: $param
     ## #set($attrib = ${transition.getAttrib($param)} )
     ## ${attrib.getType()}
#end
    /*
    * TRANSITION CODE GOES HERE.
    */
    /* Must do....
    **/
    nextState = ?;
    return nextState;
}
```

```
        #end
 #end

 /* Processed in State: STATE_Start_STA01
 * Next state: ?
 **/
public int processEvent5_invokReqPeer_Seg_Invoke_PDU(Event event_) throws
                                    NodeNotFoundException,
                                    proSPEXUsageException,
                                    TerminationException,
                                    ProNetworkSetupException  {

    /* Must set the state at end of function...
     **/
    int nextState = -1;

    Packet receivedPacket = event_.getPacket();
    Seg_Invoke_PDU segInvPDU = (Seg_Invoke_PDU)receivedPacket.getData();

    /************************************************/

    /*
    * TRANSITION CODE GOES HERE.
    */

    /************************************************/

    /* Must do....
     **/
    nextState = STATE_Start_STA01;
    return nextState;
}
```

```
}
```

Bibliography

[Ari] L. B. Arief. *A Framework for Supporting Automatic Simulation Generation from Design*. PhD thesis, University of Newcastle Upon Tyne.

[BCNN01] J. Banks, J.S. Carson, B.L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, 2001.

[BD02] F. Babich and L. Deotto. Formal methods for specification and analysis of communication protocols. *IEEE Communications Surveys and Tutorials*, 4(1):2–19, September 2002.

[BDM02] S Bernardi, S Donatelli, and J Merseguer. From uml sequence diagrams and statecharts to analysable petri net models. In *Proceedings of the Third International Workshop on Software and Performance*, pages 35–45, New York, USA, 2002. ACM Press.

[Bjo02] M Bjorkander. Graphical programming using uml and sdl. *IEEE Computer*, 33(12):17–22, December 2002.

[BK03] Morgan Bjorkander and Cris Kobryn. Architecting systems with uml 2.0. *IEEE Software*, pages 57–60, August 2003.

[BMSK95] M. Butow, M. Mestern, C. Schapiro, and P.S. Kritzinger. Sdl performance evaluation of concurrent systems. Technical report, Department of Computer Science, University of Cape Town, 1995.

[BMSK96] M. Butow, M. Mestern, C. Schapiro, and P.S. Kritzinger. Performance modelling with the formal specification language sdl. In *IFIP TC6/6.1 International Conference on Formal Description Techniques IX / Protocol Specification, Testing and Verification XVI*, volume 69, pages 213–228. Kluwer, 1996.

[BRJ98] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[CCS03] J. Hillston M. Prowse C. Canevet, S. Gilmore and P. Stevens. Performance modelling with uml and stochastic process algebras. In *IEE Proceedings: Computers and Digital Techniques*, pages 107–120. IEE, 2003.

[CLW02] A. Klemm M. Lohmann C. Lindemann, A. Thummler and O. P. Waldhorst. Performance analysis of time-enhanced uml diagrams based on stochastic processes. In *Proceedings of the third international workshop on Software and performance*, pages 25–34. ACM Press, 2002.

[CM00] V. Cortellessa and R. Mirandola. Deriving a queueing network based performance model from uml diagrams. In *Proceedings of the second international workshop on Software and performance*, pages 58–70. ACM Press, 2000.

[Con00] International Engineering Consortium. Specification and description language (sdl). http://www.iec.org/online/tutorials/sdl/index.html, 2000.

[CP03] R. Chakravorty and I. Pratt. Practical experience with http and tcp over gprs. *ACM Mobile Computing and Communications Review)*, 1(2), 2003.

[CS90] A. Chung and D. Sidhu. Experience with an estelle development system. In *Conference proceedings on Formal methods in software development*, pages 8–17. ACM Press, 1990.

[DMS03] D.W. Chadwick D.P. Mundy and A. Smith. Comparing the performance of abstract syntax notation one (asn.1) vs extensible markup language (xml). In *Proceedings of the Terena Networking Conference*, 2003.

[Dol03] Laurent Doldi. *UML 2 Illustrated*. TMSO, 2003.

[Dor02] D. Dori. Why significant uml change is unlikely. *Communications of the ACM*, 45(1):82–85, January 2002.

[Dou03] B.P. Douglass. Rhapsody for system architecture - better architecture with the uml. I-Logix Online White Paper, July 2003.

[dVHVZ96] N. de. Villiers, C. Henning, C. Vermeulen, and J. Zurcher. Sdl performance evaluation of concurrent systems 2.0. Technical report, Department of Computer Science, University of Cape Town, 1996.

[ea00] M.A. Fecko et al. A success story of formal description techniques: Estelle specification and test generation for mil-std 188-220. *The Int'l Journal for the Comp. and Telecomm. Industry*, 23(12):1196–1213, 2000.

[fS03] International Organization for Standardization. Iso 8583-1:2003 financial transaction card originated messages. ISO Online Document, June 2003.

[GB04] M. Ghanderi and R. Boutaba. Mobility impact on data service performance in gprs systems. http://itpapers.zdnet.com, 2004.

[GO03] S. Graf and I. Ober. A real-time profile for uml and how to adapt it to sdl. In *SDL 2003: System Design, 11th International SDL Forum*, pages 55–57. Springer, 2003.

[Gra02] S. Graf. Expression of time and duration constraints in sdl. In *Proceedings of the Second IEEE Sensor Array and Multichannel Signal Processing Workshop*, pages 1–16. IEEE, 2002.

[Gro02] Object Management Group. Uml profile for schedulability, performance, and time specification. Object Management Group Online Publication, 2002.

[Gro03] Object Management Group. Uml 2.0 superstructure specification. Object Management Group Online Publication, August 2003.

[HBR00] William Harrison, Charles Barton, and Mukund Raghavachari. Mapping uml designs to java. In *Proceedings of the OOPSLA 2000 Conference*, pages 178–188, 2000.

[Her03] J Herrington. *Code Generation in Action*. Manning, 2003.

[Hil01] J. Hillston. Cs4 (and msc) modelling and simulation course notes. http://www.dcs.ed.ac.uk, 2001.

[Hoe00] F Hoeben. Using uml models for performance calculation. In *Proceedings of the Second International Workshop on Software and performance*, pages 77–82. ACM Press, 2000.

[Hol91] G Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Inc98] Hyperformix Inc. Why simulate. *Capacity Management Review*, 36(2), February 1998.

[JPT00] J. Heinonen T. Oy J. Prssinen, N. von Knorring and M. Turunen. Uml for protocol engineering - extensions and experiences. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 82. IEEE Computer Society, 2000.

[Kob02] Chris Kobryn. Will uml 2.0 be agile or awkward? *Communications of the ACM*, 45(1):107–110, January 2002.

[Lit04] M.C. Little. Javasim user guide, public release 0.3. http://javasim.ncl.ac.uk/, June 2004.

[LQV01a] L Lavazza, G Quaroni, and G Venturelli. Combining uml and formal notations for modelling real-time systems. In *Proceedings of the 8th European Software Engineering Conference*, pages 196–206, New York, USA, 2001. ACM Press.

[LQV01b] Luigi Lavazza, Gabriele Quaroni, and Matteo Venturelli. Combining uml and formal notations for modelling real-time systems. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 196–206. ACM Press, 2001.

[LTB98] Sari Leppanen, Marrku Turunen, and Morgan Bjorkander. Model based design of communicating systems. Technical report, Nokia, 1998.

[Mal99] M. Malek. Perfsdl: Interface to protocol performance analysis by means of simulation. In *Proceedings of SDL Forum 99*, 1999.

[MB02] H Muhammad and M Barcellos. Simulating group communication protocols through an object-oriented framework. In *Proceedings of the 35th Annual Simulation Symposium*, pages 14–18, San Diego (New York), 2002. IEEE.

[MC01] W E McUmber and B H C Cheng. A general framework for formalizing uml with formal languages. In *Proceedings of the 23rd international conference on Software engineering*, pages 433–442. IEEE Computer Society, 2001.

[MC03] J. Merseguer and J. Campos. Exploring roles for the uml diagrams in software performance engineering. In *Proceedings of the 2003 International Conference on Software Engineering Research and Practice (SERP'03)*, pages 43–47. CSREA Press, 2003.

[MDMC96] J. Hintelmann M. Diefenbruch and B. Muller-Clostermann. Quest: Performance evaluation of sdl systems. In *IFIP TC6/6.1 International Conference on Formal Description Techniques IX / Protocol Specification, Testing and Verification XVI*, volume 69, pages 229–244. Kluwer, 1996.

[Mei02] A. Meisingset. Summary of the workshop on use of description techniques. http://www.itu.int/ITU-T/worksem/techniques/summary.html, 2002.

[MGS+00] V. Medina, I. Gmez, G. Snchez, A. Barbancho, and S. Martn. Using protocol engineering techniques to improve telecontrol protocol performance. In *IASTED International Conference on POWER AND ENERGY SYSTEMS*, 2000.

[MHSZ96] J. Martins, J.P. Hubaux, T. Saydam, and S. Znaty. Integrating performance evaluation and formal specification. In *Proceedings of IEEE ICC '96*, pages 1803–1807. IEEE Press, 1996.

[Mic88] Sun Microsystems. Rfc 1050 - rpc: Remote procedure call protocol specification. http://www.faqs.org, April 1988.

[Mic04] Sun Microsystems. Java remote method invocation specification. http://java.sun.com, 2004.

[Mil02] J. Miller. What uml should be. *Communications of the ACM*, 45(1):67–69, January 2002.

[MP00] B. Moller-Pedersen. Sdl combined with uml. In *Telektronikk 4.2000, Languages for Telecommunication Applications*, 2000.

[MTMC99] S Mitschele-Thiel and B Mller-Clostermann. Performance engineering of sdl/msc systems. *Computer Networks*, 31(17):1801–1815, June 1999.

[Ned97] AT&T & Neda. At&t & neda's efficient short remote operations (esro) protocol specification version 1.2. http://www.faqs.org, 1997.

[oSC04a] Information Sciences Institute The University of Southern California. http://www.isi.edu/nsnam/ns/ns-research.html. Research About and Using NS, 2004.

[oSC04b] Information Sciences Institute The University of Southern California. The network simulator - ns-2. http://www.isi.edu/nsnam/ns/, 2004.

[Pra04] Pragmadev. Pragmadev - real time development tools. http://www.pragmadev.com, 2004.

[Pro04] The Apache Jakarta Project. The apache jakarta project: Velocity. http://jakarta.apache.org/velocity/, 2004.

[PT00] J Parssinen and J Turunen. Patterns for protocol system architecture. In *Pattern Languages of Programs (PLoP) Conference*, 2000.

[RCP02] A. Clark R. Chakravorty and I. Pratt. Practical experience with tcp over gprs. In *Proceedings of the IEEE Global Communications Conference*, 2002.

[RHK02] M.J. Smith R.P. Hopkins and P.J.B. King. Two approaches to integrating uml and performance models. In *Proceedings of the third international workshop on Software and performance*, pages 91–92. ACM Press, 2002.

[Rou01] Jean-Luc Roux. Sdl performance analysis with objectgeode. http://www.telelogic.com, 2001.

[Ruy02] T.C. Ruys. Spin beginners' tutorial. http://spinroot.com/spin/Man/, 2002.

[Sal96] K. Saleh. Synthesis of communications protocols: An annotated bibliography. *ACM SIGCOMM Computer Communication Review*, 26(5):40–59, October 1996.

[SBD89] P. Dembinski S. Budkowski and M. Diaz. Iso standardized description technique estelle. www-lor.int-evry.fr/idemcop/uk/ est-lang/download/short-estelle-tutorial.pdf, 1989.

[Sch01] H. Schwetman. Csim19: A powerful tool for building system models. In *Proceedings of the 2001 Winter Simulation Conference*, pages 250–255. IEEE Computer Society, 2001.

[Sel03] Bran Selic. Brass bubbles: An overview of uml 2.0 (and mda). Object Technology Slovakia (OTS) 2003 Presentation (http://lisa.uni-mb.si/cot/ots2003/predkonferenca), June 2003.

[Sel04] Bran Selic. Uml 2.0: Exploiting abstraction and automation. SDTimes Website (sdtimes.com), February 2004.

[SGW94] Bran Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[Sof04] Mesquite Software. Csim19. http://www.mesquite.com, 2004.

[Spi97] *SDL* - An Annotated Specification Language for Engineering multimedia Communication Systems*, 1997.

[spi04] spinroot.com. On-the-fly, ltl model checking with spin. http://spinroot.com, 2004.

[SR03a] B. Selic and J. Rumbaugh. Using uml for modeling complex real-time systems. IBM Rational Whitepaper (http://www.rational.com), July 2003.

[SR03b] Bran Selic and Jim Rumbaugh. Using uml for modeling complex real-time systems. IBM Rational Whitepaper (http://www.rational.com), July 2003.

[Ste98] M. Steppler. Performance analysis of communication systems formally specified in sdl. In *Proceedings of the First International Workshop on Software and Performance (WOSP98)*, pages 49–62. ACM Press, 1998.

[Ste03] P Stevens. Small-scale xmi programming: a revolution in uml tool use? In *Proceedings of the workshop on XML Software Engineering 2001*, 2003.

[SW02] C U Smith and L G Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[Tec04] OPNET Technologies. Opnet modeler. http://www.opnet.com, 2004.

[Tel03] Telelogic. Introduction to telelogic tau 2.1. Help Files, 2003.

[TFKRB98] Justin Templemore-Finlayson, Pieter S. Kritzinger, Jean-Luc Raffy, and Stanislaw Budkowski. A graphical representation and prototype editor for the formal description technique estelle. In *Proceedings of the FORTE 1998 conference*, pages 37–55, 1998.

[Var04] A. Varga. Omnet++ community site and omnet++ version 2.3 user manual. http://www.omnetpp.org, 2004.

[vBB99] Jilles van Burp and Jan Bosch. On the implementation of finite state machines. In *Proceedings of the IASTED International Conference*, pages 1–7, 1999.