ANALYTICAL PERFORMANCE EVALUATION OF CONCURRENT COMMUNICATING SYSTEMS USING SDL AND STOCHASTIC PETRI NETS

A THESIS

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

DOCTOR OF PHILOSOPHY

By

Heinz Max Kabutz

February 1997

Supervised by

Prof. P.S.Kritzinger

The University of Cape Town has been given the right to reproduce this thesis in whole or in part. Copyright is held by the author.

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

© Copyright 1997
by
Heinz Max Kabutz

Abstract

In this thesis, the performance analysis of SDL with a new type of stochastic Petri net is described. This new net is called SDL-net.

The Concurrent Communicating System is described, and the need for qualitative and quantitative analysis of such systems is motivated. Formal methods are demonstrated which can be used to represent such Concurrent Communicating Systems.

The Specification and Description Language (SDL) is shown in the context of Concurrent Communicating Systems and the software development cycle is described for SDL systems. Correctness and performance of SDL are discussed and it is shown how the semantics of time for performance can be introduced into SDL by adding external information, by extending the SDL syntax or by using compiler directives. In this thesis only external information is added.

The SDL-net is introduced informally by means of a history that led to its development, and several nets are discussed that were considered for the analysis of SDL systems. How SDL may be analysed with simulation and with Petri nets is described as well as how the Coxian phase-type distribution may be used to approximate other time distributions which are necessary when modelling real systems.

SDL is then mapped to the SDL-net and the dynamic semantics of SDL are described. Three special constructs are mapped to the SDL-net by introducing special places based on the Queueing Petri net. These constructs are the signal *save* construct, the process *timer* construct and the *channel* construct and are represented by the Save-place, the Timer-place and the Channel-place respectively. Other aspects of SDL, such as the structure of SDL systems, process creating and stopping, process output, process data and process decisions are also mapped to the SDL-net and the resulting nets are described.

The SDL-net is next defined formally based on previous definitions for the queueing Petri net, and each of the new places is specified, giving the state descriptor and transitions between the states. This information can be used to derive a Markov chain for performance analysis of such an SDL-net. The order in which the transitions are enabled is also briefly described, followed by the full definition of the SDL-net and of what happens if multiple tokens arrive at a special place.

A toolset (**DNAsty**) is then depicted and it implements the mapping described in the previous chapters of the thesis. This toolset consists of a graphical SDL editor, an SDL to SDL-net converter, a tool to derive the Markov chain from the SDL-net and to solve it, and a tool to apply the results of the Markov chain analysis to the original SDL system.

The toolset is then applied to the *InRes protocol*, which is an extension of the *alternating-bit protocol* and is often used as a benchmark for SDL toolsets. It is shown how a deadlock can be detected using the SDL-net, and this is confirmed using a commercial SDL simulator. The performance of the InRes protocol is measured with the SDL-net and the results interpreted and discussed.

This thesis then concludes that it was indeed possible to analyse SDL systems with this new SDL-net and proposes some areas of possible further research.

Acknowledgements

During the course of this study numerous people have assisted me in various technical and non-technical ways. I would like to thank the following people in particular, however, for making this thesis possible:

- My Lord Jesus Christ for inspiring me through his Word to work diligently at whatever I do and to give thanks in all circumstances. Without **Him** I would not have been able to do this thesis.
- My supervisor Professor Pieter Kritzinger, the best supervisor I could have chosen, who kept on encouraging me, always showing a keen interest in my work and always ensuring that I had enough finances to survive.
- My lovely and beautiful wife Hélène who has supported me selflessly during the writeup of this thesis and beyond. I thank you too for loving me and praying for me continuously.
- My parents Hans und Sigrun for encouraging me to study as extensively as possible.
- Peter Kemper and Falko Bause from the University of Dortmund for their help during the initial phases of this thesis and for letting me use their Markov chain analyser USENUM and the QPN-Tool.
- John Green and Stephen Donaldson for their mathematical and theoretical input.
- William Knottenbelt for the use and support of DNAmaca, and for letting me use some code of DNAnet for invariant analysis. Your good humour and excellence in all you do have often inspired me.
- My colleagues from the Data Network Architectures Laboratory, particularly Mark Mestern and Conrad Vermeulen for many interesting discussions revolving around the subject of this thesis.
- Vincent Encontre, Rodolphe Arthaud and other staff at Verilog for many interesting and informative discussions about SDL during my working at the company.

- Johann van der Walt and Plessey Tellumat Ltd. for giving me the opportunity to spend 3 months as part of a project team designing a new type of telephone switch using SDL.
- My friends Andrew Righthouse, Edith Sher, Henry Jackson and the late Irma Boyce who through their constant praying kept me going. May your prayers be answered!
- My friends John Green, Anton de Swardt, Richard Sharpley and Jean van Eeden for being great friends and convincing me to continue with my studies.
- My HUGE family consisting of siblings: Bettina, Rudolf and Marten and in-laws (not out-laws): Gregory, Anestis and Costas Kytides, Gillian and Cameron Clark, Lorraine and Abigail Jones and Karin Kabutz for being so kind and looking out for us always.
- The staff at the department of computer science for many interesting debates about various aspects of computer science. Thanks also to Dr. Mike Linck for helping to print out "Word for Windows" documents; a special thanks also to the administrative staff Peter Fortuin, Mary Wood, Eve Gill and Rowena Evans for always being glad to help me even when it was inconvenient for them and thanks to Aleks Strez and Sandi Donno for keeping the computers humming at top speed.
- The South African Foundation for Research Development (FRD) for funding this research.

Contents

Acknowledgements **ii**

1 Concurrent Communicating Systems with FDT **1**

2 Specification and Description Language (SDL) **3**

2.1 Overview of SDL 3

2.2 SDL in the Development Process 4

2.3 Correctness and Performance 5

2.4 Semantics of time in SDL for performance 6

2.5 How is time introduced? 7

2.5.1 External information 7

2.5.2 Extending Syntax 8

2.5.3 Compiler Directives 8

3 SDL-net **9**

3.1 History 9

3.2 Analysing SDL with the SDL-net 10

3.2.1 Correctness Analysis with SDL-net 12

3.2.2 Performance Analysis with SDL-net 12

3.2.3 Summary 13

3.3 Other Methods of Analysing SDL 13

3.3.1 Simulation 13

3.3.2 Petri nets 15

3.3.3 Coxian distribution 16

4 Mapping of SDL to SDL-net 23

4.1 Mapping SDL in general 23

4.1.1 Dynamic semantics 23

4.1.2 Extended Finite State Machines 25

4.2 Special constructs 27

4.2.1 SAVE Construct 27

4.2.2 TIMER 31

4.2.3 CHANNEL 35

4.3 Other SDL constructs 39

4.3.1 Structure of SDL Systems 40

4.3.2 Process Create and Process Stop 43

4.3.3 Process Output 45

4.3.4 Process Data 48

4.3.5 Process Decision 52

4.4 Summary of Chapter 53

5 Definition of SDL-net 55

5.1 Basic Definitions 55

5.1.1 Coloured Petri net 55

5.1.2 Generalized stochastic Petri net 56

5.1.3 Coloured generalised stochastic Petri net 56

5.1.4 Queueing Petri net based on CGSPN 57

5.2 New Queued Places 58

5.2.1 Save-place 58

5.2.2 Timer-place 60

5.2.3 Channel-place 62

5.3 SDL-net 65

5.4 Conclusion 66

6 DNAsty 69

6.1 SDLite 71

6.2 SDL API and Sdl2pn 72

6.3 Wam2mod 74

6.4 DNAmaca 75

6.5 Mod2sdl 77

6.6 Conclusion 79

7 Application of SDL-net 81

7.1 InRes Protocol 81

7.2 Correctness 88

7.2.1 Deadlock 88

7.2.2 Order of Signals affecting Correctness 90

7.3 Performance 91

7.3.1 Experiment description 91

7.3.2 Results 92

7.3.3 Conclusion 98

8 Conclusion 101

A Mathematical definitions of nets 103

A.1 Nets without time 103

A.1.1 FIFO queue net (FIFO net) 103

A.1.2 Predicate-Transition net (Pr/T net) 104

A.2 Nets with time 104

A.2.1 Time Petri net (TPN) 104

A.2.2 Guard Time Petri net (GN) 105

B InRes Protocol Described in SDL/PR 107

Bibliography 113

List of Figures

3.1 Case tools produced by the DNA-Lab for analysing Concurrent Communicating Systems 10

3.2 Analysis path of SDL using Petri nets. 17

3.3 Exponential distribution. 18

3.4 Erlang (hypo-exponential) distribution. 18

3.5 Parallel phase (hyper-exponential) distribution. 19

3.6 Coxian phase distribution. 20

4.1 Overall structure of Dynamic Semantics Interpretation Model 24

4.2 Structure of Dynamic Semantics Interpretation Model for SDL Process Instance Set 24

4.3 Example of valid process states 26

4.4 Expanded process of Figure 4.3. 28

4.5 SDL-net mapping of Figure 4.4. 28

4.6 SDL process using the save construct. 29

4.7 State and Signal queue changes with Save construct. 29

4.8 A Save-place and its shorthand notation 32

4.9 SDL Process where implicit input is not expanded. 32

4.10 SDL Process with implicit input for signal **s2** expanded. 32

4.11 SDL-net mapping of Figure 4.10. 33

4.12 A Timer-place and its shorthand notation 33

4.13 "Reset" net-construct for timer t. 36

4.14 "Save" net-construct for timer t. 36

4.15 "Set" net-construct for timer t. 37

4.16 Additional edges for timer signal input. 37

4.17 System with both immediate and delayed channel 38

4.18 A Channel-place and its shorthand notation 41

4.19 Equivalent generated SDL-net of Figure 4.18. 41

4.20 SDL hierarchy. 41

4.21 Hierarchy of a simple PABX specified in SDL 42

4.22 SDL-net of PABX structure 42

4.23 SDL Processes with different initial and maximum instances 44

4.24 SDL-net of **PROCESS p1(0)** and **PROCESS p2(0,1)**. 46

4.25 SDL-net of **PROCESS p3(1)** and **PROCESS p4(2,3)**. 46

4.26 Two outputs in one transition.. 47

4.27 SDL-net where order of the two outputs is lost. 47

4.28 SDL-net where order of the two outputs is preserved. 49

4.29 SDL Process containing variables. 49

4.30 SDL-net of Figure 4.29. 50

4.31 SDL-net equivalent of **count := count + 1**. 50

4.32 SDL-net equivalent of **count := 3**. 51

4.33 SDL-net equivalent of **count** < **3**. 51

4.34 SDL Process with non-determinstic (**any**) decision and deterministic (**count**) decision. 52

4.35 SDL-net equivalent of Figure 4.33. 53

4.36 This diagram illustrates which SDL constructs are supported by this analysis method. 54

5.1 A queued place and its shorthand notation 57

5.2 Transitions in the Save-place 60

5.3 Transitions in the Timer-place 63

5.4 Transitions in the Channel-place 67

6.1 Toolset implemented to verify SDL-net analysis technique. 70

6.2 SDLite used to specify a SDL/GR system 71

7.1 InRes system level view 83

7.2 InRes Initiator block level view 83

7.3 InRes Initiator process level view 84

7.4 InRes User Initiator process level view 85

7.5 InRes Responder block level view 85

7.6 InRes Responder process level view 86

7.7 InRes User Responder process level view 86

7.8 InRes Medium block level view 87

7.9 InRes Medium process level view 87

7.10 Sequence of messages leading to a deadlock. 89

7.11 SDL-net demonstrating that the order of signals may affect the correctness of the system. 90

7.12 Mean time Initiator process spends during **experiment 1** waiting for a Connection Confirm (CC) signal. 93

7.13 Mean time Initiator process spends during **experiment 2** waiting for a Connection Confirm (CC) signal. 93

7.14 Mean time Initiator process spends during **experiment 1** waiting for a Data (IDA-Treq) signal or a Disconnect (DR) signal. 94

7.15 Mean time Initiator process spends during **experiment 2** waiting for a Data (IDA-Treq) signal or a Disconnect (DR) signal. 94

7.16 Mean time Initiator process spends during **experiment 1** waiting for a Data (IDA-Treq) signal or a Disconnect (DR) signal. 96

7.17 Mean time Initiator process spends during **experiment 2** waiting for a Data (IDA-Treq) signal or a Disconnect (DR) signal. 96

7.18 Mean time Initiator process spends during **experiment 1** waiting for an Acknowledgement (AK) signal. 97

7.19 Mean time Initiator process spends during **experiment 2** waiting for an Acknowledgement (AK) signal. 97

7.20 Time taken to transfer a file from the Initiator to the Responder during **experiment 1**. 99

7.21 Time taken to transfer a file from the Initiator to the Responder during **experiment 2**. 99

List of Tables

6.1 Example of delays for SDL actions. 74

7.1 Process states when deadlock occurred. 89

Chapter 1

Concurrent Communicating Systems with FDT

From the early stages of computer software and hardware design it has been critical that systems perform correctly and efficiently. This is becoming more important as computer systems become more distributed and as the complexity of the tasks increases. When the failures of such systems can lead to the loss of human life and to the destruction of irreplaceable information and property, such errors can then no longer be overlooked. Examples of such errors are listed by Neumann in [Neu, Neu95], one of these was the aborted NASA space mission in 1975, caused by a deadlock in the control system, another was the shutdown of a large part of the AT&T switching network in 1990 due to a synchronisation error in the routing software. From hardware to software design, methods are needed to automatically verify the systems and to ensure that the response time will be adequate.

These distributed systems as labelled Concurrent Communicating Systems. These include network protocol implementations, traffic control systems and software for the control of medical systems. For such critical systems, an *ad hoc* approach to software design is neither desirable nor sufficient, therefore a more formal approach is necessary.

Ambiguity in the system specification must be avoided and this is possible by using a *formal method* to describe the system rather than a natural language. Such a *formal method* can then also be used as a basis to automatically generate test cases and to analyse the correctness and performance of the system.

One such formal method is to use a Formal Description Technique (FDT) which can be used to specify and describe systems; examples of FDT's based on finite state machines are the Specification and Description Language (SDL) [OFMP+94, IT93b] and Estelle [Hog89] and a further FDT based on process algebras would be LOTOS [Hog89]. The Petri net invented by Carl Adam

Petri in 1962 [Pet62], is another formal method for describing concurrency and synchronisation inherent in Concurrent Communicating Systems.

One way of analysing correctness is by exploring the state space. This is done by generating the reachability tree containing every state and the transitions between the states. This, invariably large, state space may be stored using a probabilistic bit state hashing as first used by Holzmann [Hol91] or more recent improved methods [Kno96]. A state space explosion problem may occur when the number of unexplored states increases significantly faster than the rate at which they are explored. Partial state space exploration methods are then often used to *test*, rather than to *prove*, whether a system is correct. If the Concurrent Communicating System could somehow be represented as an ordinary Petri net, one could use analytical techniques such as *place* and *transition invariant analysis* to avoid the state space explosion problem. An ordinary Petri net is however a major abstraction of the Concurrent Communicating System and the results may sometimes not relate realistically to the original system, particularly when the order of signals in the Concurrent Communicating System is not represented in the Petri net abstraction and time is involved.

Performance analysis, when done at all, is usually done by simulating the specification once it has been translated to some executable code; either real code or code for a virtual machine [BMSK96]. In order to attain a reasonable level of confidence in the results, such simulations may require a lot of computer time and this is not ideal either. Another way to analyse the performance is to represent the Concurrent Communicating System as a Stochastic Petri net which easily translates to a Markov chain. Stochastic Petri nets are designed to be used with negative exponentially distributed time and may not be suitable for representing other time distributions often found in real systems.

Once the system has been specified in an FDT, and this has been analysed for correctness and performance, the system should be implemented; ideally the FDT specification becomes the basis for such an implementation. An example of such a mapping from a FDT to an implementation is found in [Ver96a] where C code is generated from SDL.

A lot of other work has been done in determining the correctness and performance of Concurrent Communicating Systems described in a FDT; this will be discussed in the appropriate places in the remainder of the thesis.

The rest of this thesis introduces SDL and shows how performance semantics can be included in the language. Various methods of analysing SDL systems are discussed, and a new net is defined to determine the performance of Concurrent Communicating Systems described in SDL. A mapping from SDL to this new net is presented, followed by a formal definition of the net. The toolset designed to analyse Concurrent Communicating Systems with this new net is discussed and its application is illustrated with the SDL specification of the InRes network protocol.

Chapter 2

Specification and Description Language (SDL)

An FDT which has found a great deal of support in industry for the specification from network protocols through embedded software systems is SDL. SDL covers the whole software development cycle from specification to the description of a system. It is natural to start with SDL and provide ways of directly testing correctness and performance during the specification cycle of the Concurrent Communicating System. Ways are discussed of introducing aspects of performance into a SDL description either by augmenting the syntax or specifying the required information external to the specification. In this thesis, the 1992 version of SDL, namely SDL-92, is used with the exception that types and packages are not supported.

2.1 Overview of SDL

SDL is a widely recognised international standard maintained by the ITU-T. It was first formalised in 1976 at which stage it only contained recommendations on how to draw process graph symbols. In 1980, SDL was revised and the block structural concept introduced. It was at this time that first steps were made to also define a textual equivalent of the up to now graphical SDL. In 1984, abstract data types were introduced which was later built upon to produce SDL-88. The latest big revision occured in 1992 and involved introducing object oriented extensions to SDL.

Since SDL has been used now for over 20 years, it has reached relative stability and is widely accepted internationally. There are tens of thousands of users in more than 1000 companies worldwide. Users of SDL include the European Telecommunications Standards Institute (ETSI), AT&T, Ericcson, Alcatel, Renault, BT and many others. The graphical SDL editor SDLite produced as part of this thesis has been distributed to hundreds of users worldwide. One of the reasons for the

popularity of SDL is that it is a visual language that provides the user with a good overview of the system being designed. The recent Object Oriented extensions provide an ability to integrate SDL with traditional OO techniques, making it even more attractive for system designers.

SDL is ideal for design interactive real-time systems which communicate via signals. SDL was originally used by telecommunications companies for highly complex system development. SDL is also effective for developing distributed systems where synchronisation issues become extremely complex. Developers of safety-critical systems employ SDL as it can ensure that a design will work before implementation.

SDL is a formal specification language that allows its users to produce a thorough system specification and design and prove the design will work before going through the cost of implementing the system. SDL has a static and dynamic structure. The static structure is defined in terms of blocks and channels which represent the architectural structure of the system. The dynamic structure is defined by processes and signal routes. Processes define the dynamic behaviour of the processes using finite state machines. SDL uses data encapsulation rather than global data so that processes only can share data by sending over a signal route. Upon receiving a signal, a process typically moves from one state to the next. Processes are dynamic and can be created and stopped during the system life cycle.

It is beyond the scope of this thesis to include a more detailed discussion on SDL and how it works. The most accurate reference of how SDL works is the Z.100 [IT93b] standard. There are also books for learning SDL such as [BHS91], [BH93] and [OFMP+94].

2.2 SDL in the Development Process

The key success of a system is its thorough specification and design. This requires a suitable FDT which can provide amongst others [IT93d]:

- a well defined *set of concepts*; and
- *unambiguous, clear, precise* and *concise* specifications; and
- a basis for *analysing* specifications for *completeness* and *correctness*; and
- use of computer based *tools* to create, maintain, analyse and simulate specifications.

A formal specification is not intended to run on a computer but is merely an abstraction that can serve as a basis for implementation and should abstract from implementation details in order to give a better overview of a complex system and to postpone implementation decisions. The formal

specification can also be used to simulate and analyse the design which can reveal critical design errors early in the software development life cycle.

When a system is represented only in terms of its behaviour from the outside, then it is referred to as a *specification*. When the internal structure of the system is also represented, then it is referred to as a *description*. SDL allows both the specification and the description of systems.

SDL is mainly used in the telecommunication industry but can also be used to describe systems in other areas [IT94] where the *type of system* is real-time, interactive or distributed, where the *type of information* concerns of the behaviour and structure of the system and the *level of abstraction* ranges from overview to detail.

In this thesis only those systems that have been described formally will be considered for automatic correctness and performance evaluation and not those that have been described informally.

A more complete investigation of the use of SDL in the software cycle is given in [IT93d, IT94].

2.3 Correctness and Performance

In most software, one of the main criteria for its success is whether it works "correctly", especially in the critical application areas in which SDL is used. Usually "correctness" is validated according to the original design for the software. However, there are some criteria that apply to software regardless of the application area. These can then be verified automatically without referring to the original design. Some of such correctness criteria include:

Boundedness Even though the language SDL is not limited by physical resources, all our current computer systems are. A system is then incorrect when some part of it tries to use an infinite number of resources. An example in the SDL context is the signal queue which potentially could contain an infinite number of signals but which should be bounded in any real system. For system design it would be useful if one could determine the optimal number of resources needed for the system.

Deadlock The system should not get into a state where two or more processes are waiting for each other to do something before each can proceed. In SDL, two processes may wait to receive signals from each other before continuing. This would result in the system having a deadlock.

Home State It is sometimes useful to know whether the system has a state or states to which it will return from every other state. This is particularly necessary if a Markov chain is derived to analyse performance.

Dead Code It can happen that some parts of the system can never be reached during an execution. Even though this would usually not result in an incorrect system state, it would be beneficial to reduce the size of the system by eliminating the dead code where possible.

Often performance characteristics of systems are considered only as afterthoughts. It is usually more important to the designer that the system be correct rather than that it performs well enough. In time critical systems, such an approach is no longer acceptable. Again, such performance can be measured subjectively, if it "feels" fast enough, or objectively, producing information about the efficiency of the system before implementation.

Time spent in state In SDL, it can be useful to look at how much time each process spends in its various states. This can be used to determine where in the process the bottlenecks are and what parts should be more streamlined.

Throughput The throughput of a system is used as a measure to calculate how quickly a system will respond, such as when data is transferred across a network. It can indicate parts of the system that cause unnecessary delays in the system.

Average Queue Length The average length of the signal queue will determine the time delay before a signal is serviced. By increasing the speed of a signal server, an improvement in the performance of the signal transfer can be achieved. Performance values such as this can indicate which servers are critical so that these may be targeted for performance improvement.

Since SDL does not include semantics of time that can be used for performance evaluation, it is necessary to first specify these.

2.4 Semantics of time in SDL for performance

A fundamental difference between the SDL system and the implementation is that the SDL system is not limited by processing resources. It is assumed that the system is always fast enough. This is not always realistic, in that each process transition and each signal transfer will take some time and use some system resources. Processes that execute on different machines might also differ in processing speed which will affect the relative speed of the system. It is thus not possible to calculate the performance of a SDL system without introducing semantics for time.

The method to be discussed in this thesis associates a delay with each SDL action, used to calculate the delay for a process input transition depending on the actions performed in the transition and

their specified delays. Other performance characteristics include the delay of signals transferred between processes and the relative length of process timers. Each of these time delays are set relative to each other so that a realistic value for a timeout is calculated in terms of process action and channel delays.

Bause et al [BB90] proposed a timed version of SDL-88 (TSDL) where exponential rates are assigned to process transitions to model the delays occured by the transitions. The user has to explicitly specify the rate of each SDL transition, which would be difficult and time-consuming for larger systems. When a transition is changed, the rate has to be re-calculated manually.

Another method of specifying performance measures came about recently with the development of Queueing SDL (QSDL) in [DHHMC95] which uses additional queueing constructs to model delays in process transitions and resource contention between processes. The model is simulated by mapping the SDL model to C++ and executing that. No analytical methods seem to be used to determine the correctness and performance of the SDL system. Since this is a new method, it is still to be seen how practical it is in representing realistic protocols.

2.5 How is time introduced?

Several methods exist for introducing time into a SDL system. They range from using external information to extending the syntax of SDL, to adding compiler directives, as will be explained. External information is used in this thesis, because existing tools can then be used to analyse and design the system. It also allows the performance values of a system to be varied easily without having to change the original SDL specification and can so be used to produce a range of performance results from the same SDL specification.

The actual time values for the system can be obtained in a number of ways. For example, one could try estimate how long each action should take and from that derive the delays of the transitions. One could also execute an implementation of the system and measure the time needed for each transition. In our approach, we choose to estimate how long each action should take.

2.5.1 External information

The method used in this thesis does not change the syntax of the Z.100 standard language SDL but stores the additional timing information in supplementary external control files. These contain the delay of each action in SDL, the time delay distribution of a timed channel and the duration of a timer. In addition, the performance information collected is filtered using a control file, so that only the required results are displayed.

SPECS in [BMSK96] uses a similar method of control files to specify information such as channel delay, channel reliability, process speed and block speed.

2.5.2 Extending Syntax

The syntax and semantics of SDL/PR are extended in TSDL [BB90] to represent the performance information such as the rate of each SDL transition. In addition, the tool used to evaluate TSDL systems requires a control file that contains additional information necessary to represent TSDL internally. TSDL was only established for the textual representation (TSDL/PR) and no graphical equivalent (TSDL/GR) was defined. The problem with this method is tool support for the parsing and editing of TSDL and future versions of SDL might require that TSDL again be extended.

Similarly, the syntax and semantics of SDL are extended for QSDL [DHHMC95] to include machines used to model timing information. QSDL is defined for both the graphical and the textual representation, but no graphical editor for QSDL/GR exists. The same criticism as for TSDL applies, in that existing tools will not be able to process QSDL.

2.5.3 Compiler Directives

Another method of representing time is shown in PEW [WK92, Whe93], a toolset used to analyse Estelle systems. PEW uses compiler directives to represent performance values by specifying these directives in comments. This is similar to the SDL/GR Common Interchange Format (CIF) [IT96] used to exchange formatting information between graphical SDL editors. Such systems can still be validated using existing parsers, but a compiler needs to be written to parse the information contained in the comments.

Chapter 3

SDL-net

This chapter will describe how SDL is analysed, focusing specifically on the technique proposed in this thesis, that is the SDL-net. The SDL-net is a new stochastic net designed to model systems specified in SDL by concentrating on analysing their performance. This chapter shows the underlying basis for the SDL-net and gives a brief history. It also demonstrates how the existing analysis techniques for queueing Petri nets may be applied to analyse the newly created SDL-nets.

3.1 History

The idea of mapping a system specified in an FDT to a stochastic Petri net was thought of in 1993 when, as part of an advanced course on networks, parts of the LAP-B protocol were specified using a generalised stochastic Petri net while other students used Estelle for the same specification. In the same year, it was also attempted to specify the DQDB protocol using Petri nets. In both cases it was realised that Petri nets are a too low-level formalism and it was far easier to use a high-level language like Estelle. However, while many tools existed for Petri nets, this was not the case with Estelle.

In 1994, research work was started to develop two *Computer Aided Software Engineering* (CASE) toolsets, illustrated in Figure 3.1, one for Petri nets and the other for SDL. Additional research was started, of which this thesis is a product, on mapping systems specified with SDL to Petri nets. A more detailed description of the exact building blocks produced in this thesis will be shown in Chapter 6.

An investigation revealed that several groups had mapped SDL to various types of Petri nets, such as the ordinary Petri net, the FIFO net, the time Petri net, etc. However, with these nets, it was very cumbersome to represent the complex queueing strategies used by SDL to determine which signal would next be accepted by a SDL process. The purpose of these mappings was to

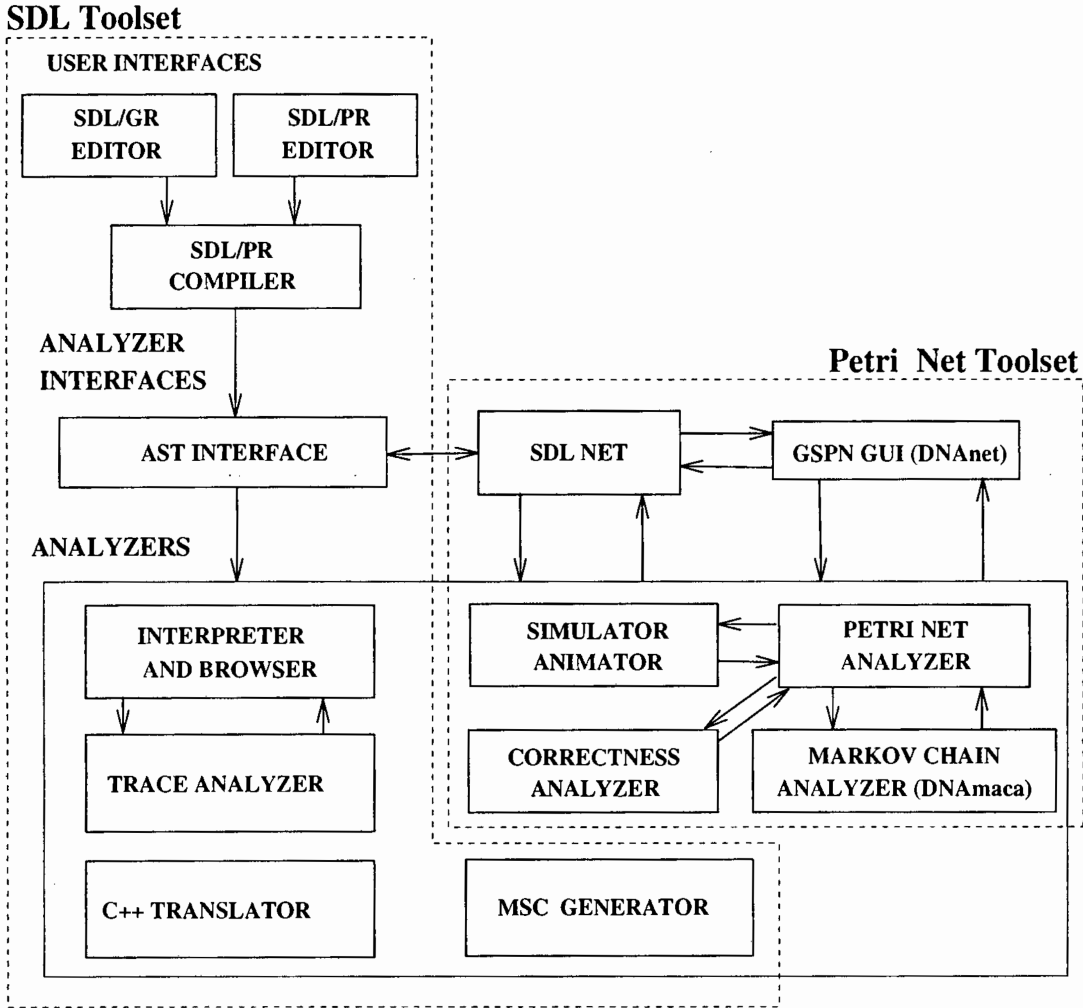

Figure 3.1: Case tools produced by the DNA-Lab for analysing Concurrent Communicating Systems

analyse the functional correctness of the SDL system using net mappings and not its performance characteristics. It was decided that a special net was necessary to represent the queueing strategies used by SDL - this net would have to be a stochastic Petri net so that the performance of an SDL system could be analysed.

A SDL system was mapped manually to such a SDL-net to ascertain the feasibility of an automatic mapping. A modified version of QPN-Tool [BK91] was used which enabled one to analyse some aspects of the SDL-net, especially the performance of the system [FKPP95].

3.2 Analysing SDL with the SDL-net

This thesis is the first that uses Petri nets to analyse the performance of SDL systems. Many previous projects have investigated the use of nets for analysing the correctness of SDL [Gra90, TFD94,

KMT88, KCYH91], and some of those techniques may be used to determine the correctness of SDL systems before calculating the performance.

In order to make use of the analytical methods for Petri net analysis and Markov chain analysis, a net was needed that would fulfill the following requirements:

- The net should not be too low-level as it should retain as much as possible of the SDL semantics.
- For the performance analysis a stochastic timed net was necessary from which the Markov chain could be derived. In addition, it should be possible to represent time with the Coxian distribution, discussed in Section 3.3.3 on page 16.
- It should be relatively easy to represent some more complicated SDL constructs such as queues and timers.
- Existing techniques for analysing correctness should still apply to the net where possible.
- Where possible, existing methods and tools should be used.

The lowest level of Petri net considered in this mapping was the coloured Petri net (CPN), as will be defined in Section 5.1.1, this assigns colours to tokens and existing methods can then be used to flatten the net to an ordinary Petri net. Existing Petri net techniques, such as the calculation of place and transition invariants, are used to analyse the flattened net for correctness. A disadvantage of using this CPN is that it is still too low-level for the representation of complicated data types, therefore an Algebraic Petri net (APN) [Rei91, DH90] could have been more appropriate. Since the main focus of this thesis is on the modelling of performance and not on modelling complex data structures, the CPN was chosen as the basis for the analysis.

In addition to using tokens with different colours, it is also necessary to include stochastic time. The generalised stochastic Petri net (GSPN), as will be defined in Section 5.1.2, allows certain transitions in the Petri net to be timed and others to be immediate. The GSPN can easily be mapped onto a Markov chain, but, as it is based on the ordinary Petri net, it is too low-level for adequately representing the different signals sent between processes of a SDL system. A coloured GSPN (CGSPN) was chosen, as will be defined in Section 5.1.3, which combines the timing information of the GSPN with the coloured tokens of the CPN.

One difficulty in modelling systems with Petri nets is that queues cannot be represented easily, especially for more complicated queueing strategies with varying priorities for customers. The queueing Petri net (QPN) [Bau93], defined in Section 5.1.4, is a net based on the CGSPN that introduces queued places to model queues. The queues have to be defined in such a way that

no tokens are created or destroyed during the scheduling of the queue. Common queues defined for the QPN are FIFO, LIFO, Coxian with processor sharing and Coxian with infinite servers. These queues are not expressive enough to represent some of the more complex queueing strategies found in SDL. Another net that is used to represent queues is the FIFO-net as described in Appendix A.1.1 on page 103. This is not suitable for analysing performance as it does not include semantics for time.

It was decided to define a new net, called the SDL-net, based on the QPN. Three new places are defined that are used specifically to represent certain aspects of a SDL description which are important for performance evaluation. The first is the Save-place which models the signal queue in the SDL process and includes support for the *save* construct that dynamically re-orders the signals in the queue. The second is the Timer-place which is used to model the deterministic process *timer* where the service can be interrupted when the timer is reset. The third is the Channel-place which models the delay of signals being transferred across a *channel* between communicating systems.

3.2.1 Correctness Analysis with SDL-net

When analysing the correctness of a QPN, a queued place is considered as an ordinary place and existing techniques are used to analyse the Petri net. Invariant analysis techniques are used on the underlying Petri net to obtain results for the QPN. It is shown in [Bau93] that the QPN is bounded when the underlying Petri net is bounded. Should the invariant analysis fail to reveal the necessary results, then the coverability graph of the underlying Petri net is calculated. If this also does not produce the required results, then the state space of the QPN is explored.

These efficient analysis techniques available for the QPN could be applied to the SDL-net but it had to be ensured that tokens are preserved in the queued places.

These three methods are listed in order of their efficiency, where the invariant analysis is the most efficient and the state space exploration of the SDL-net the least efficient. It should be noted that state space exploration of the SDL-net is the most general method covering all possible SDL-nets. This is used only if the other methods fail to determine that the system is correctness.

3.2.2 Performance Analysis with SDL-net

The performance of a SDL-net can be determined by simulation or by solving the derived Markov chain. It was chosen to derive the Markov chain from an SDL-net using existing techniques and special transitions for the new places. The Markov chain is constructed based on information obtained during the correctness analysis of the SDL system. Specifically, the maximum number of signals in the signal lists needs to be known to construct the queues in the Markov chain.

The parameters for the Markov chain are determined by the rates of the timed transitions and by the weights of the immediate transitions in the SDL-net. For the queued places, the rates are determined by the parameters specified for the Coxian distribution as further explained in Section 3.3.3 on page 16. The rate of each transition in the SDL-net should reflect the amount of time that the SDL system would spend in that transition and is calculated according to the SDL actions that need to be performed during that transition.

3.2.3 Summary

To summarise, the SDL-net is a new net that is based on a coloured generalised stochastic Petri net with queued places to model certain SDL constructs, for which existing methods are used to analyse the correctness and the performance of SDL systems.

3.3 Other Methods of Analysing SDL

Some aspects of simulating SDL are discussed in this section, referring specifically to the automatic analysis of the functional correctness of the system and its performance. Previous work in determining the correctness of SDL systems using Petri nets is discussed, and a new method for determining the performance of such systems is presented using the SDL-net.

3.3.1 Simulation

The most common method of analysing Concurrent Communicating Systems described in SDL is by deriving a target implementation that is executed on either a virtual machine or on a real system.

Care has to be taken when a virtual machine is used that the results still apply to an implementation for a specific target machine. The advantange with using a virtual machine is that more general, implementation independent information is derived for the specification. It is also often easier to collect the data from within a virtual machine than from within a real processor, because tracing functionality is usually built into the virtual machine.

Another method is to represent the model in a language such as C++ [Str91] that can be compiled on a real system and used as a basis for simulating the model. This model is then also the basis for implementation, resulting in more implementation specific and more accurate results as is done in the SDL-toolset of Figure 3.1. A disadvantage is that the results cannot always be easily applied to another target architecture.

Correctness Analysis with Simulation

The target implementation is used to simulate the execution of the model and can in this way be used to automatically test for certain correctness measures such as those already listed in Section 2.3 on page 5. In addition, the execution of the model can be compared to a functional specification of the system with Message Sequence Charts [IT93c] to verify that the SDL system corresponds to what is functionally expected.

An advantage of using simulation is that even very large systems can be simulated and *some* results obtained, however, because the simulation cannot guarantee that all states are explored, these systems may still not be correct. A simulation that runs for a limited number of time will only reveal information about what happens during that time and cannot *prove* the correctness of the system.

An example of a SDL simulator that maps the target implementation onto a real system is the *Object***GEODE** simulator [Ver96b] which maps the SDL system onto **C** code. The simulator can step through the execution of the system, set break-points and watches on data variables and generate execution traces as message sequence charts. The **C** code that is generated with *Object***GEODE** may be used as a basis for implementation.

An example of a simulator that maps the target implementation onto a virtual machine is the PEW toolset [Whe93] used to analyse systems specified in the FDT Estelle. The Estelle system is mapped onto a machine code understood by the virtual machine with the advantage that certain aspects of concurrency are analysed more accurately than if it were executed on a real processor. The disadvantage with this method is that the overhead of interpreting the virtual machine code is quite high and this can slow down the analysis significantly.

Performance Analysis with Simulation

Similarly, simulation will usually return *some* performance results but these may not accurately reflect what will happen throughout the life-span of the system. For example, it might take a relatively long time for the system to reach a steady state. If a simulation is not executed for long enough, the transient stage may skew the performance results of the overall system.

The SPECS [BMSK96] toolset simulates the target implementation by means of a virtual machine. The SDL system is represented as machine code that, when executed on the virtual machine, generates trace data which is parsed to determine the performance of the system. This method is rather slow because of the overhead of interpreting the machine code in the virtual machine.

The QUEST [DHHMC95] toolset maps queueing SDL (QSDL) systems onto the **C++** programming language which is simulated to obtain performance data. Such data may not easily be applied

to another architecture as it may be machine and implementation specific and may depend on the architecture on which it was generated. The advantage with generating a programming language is that the execution is very fast and that the results will apply very accurately to at least the processor architecture.

3.3.2 Petri nets

Another method of analysing Concurrent Communicating Systems is by mapping them onto some Petri net which is analysed using known techniques. Efficient analytical techniques such as the calculation of *place invariants* and *transition invariants* are utilised for Petri net analysis to prove the correctness of the Concurrent Communicating System. Since the net is, similarly to the virtual machine, usually an abstraction or model of the system, care needs to be taken when applying the results of the Petri net analysis to a real system. The advantage of using nets is that the theoretical basis of nets is very well established and many techniques exist to analyse nets and to prove their correctness. In addition, a Markov chain can be derived from a stochastic net which, when solved using existing techniques [Kno96], yields performance results.

A lot of work has been done in proving the correctness of SDL systems with nets, but, up to now, nets have not been used to evaluate the performance of SDL systems. This may be due to the complexity and size of the resulting nets, as some SDL constructs can only be represented with great difficulty using ordinary Petri nets.

Correctness Analysis with Petri nets

Extensive work has been done in testing the functional correctness of SDL specifications using various classes of Petri nets. Some Petri net classes included timing information, but merely to test the correctness of the system with regard to timers.

In 1990, Jens Grabowski [Gra90] mapped SDL-88 to several classes of untimed Petri nets, namely the Place/Transition net (P/T net), the FIFO-net, defined in Appendix A.1.1 on page 103 and the Predicate/Transition (Pr/T net), defined in Appendix A.1.2 on page 104. As these nets do not include time, they are not suitable for analysing performance. The mapping of the P/T net did not take into account the order of the signals arriving at the process nor the values of the process variables. However, efficient analysis techniques could be applied to the P/T net to test the correctness of the system. The mapping to the FIFO net took into account the order of signals, but no efficient techniques were given for analysing the net. Certain SDL constructs depending on data variables were mapped onto an Pr/T net but again, it was not clear how efficient analysis techniques could be used.

Extensive work is being done at the Humbolt-University of Berlin in mapping SDL-92 to a Time Petri net (TPN) [Leh93, Tau93a, Tau93b, TFD94], defined in Appendix A.2.1 on page 104. The TPN has an earliest-firing-time (*eft*) and latest-firing-time (*lft*) on certain transitions so a deterministic timing delay is modelled by setting the *eft* and the *lft* to the same time. The Guard Time Petri net [Tau93b], defined in Appendix A.2.2 on page 105, is an extension of the TPN and may be used to specify the order of signals in the queue. It has guards associated with the transitions to determine whether the transition should be enabled. Even though time was used in the modelling, the aim was to verify the functional correctness of the SDL system more accurately by considering the process timers, and no performance evaluation is done. Furthermore, no efficient techniques are given for analysing the TPN and only state space exploration is possible.

In 1991, Kim *et al* [KCYH91] mapped SDL to a numerical Petri net which is a coloured Petri net with additional read/write memory. The additional memory represents process variables to calculate which decision branch a transition should take. It seems from the literature [KCYH91] that SDL tasks, which are used to assign new values to variables, are ignored in the mapping, so decisions that depend on the values of variables will still not be accurately mapped as the value of the variables cannot be changed from within the Petri net model. The purpose of this mapping is again to test the correctness of SDL systems and no performance evaluation is possible with this method.

Performance Analysis with Petri nets

This thesis presents the first method of performance analysis of SDL with a Petri net. In any analysis it is important to map the results back to the original system. For example, if a Petri net is used to model the X25 protocol [Bil94], the results of analysing the Petri net should reflect what happens in the real X25 system. Figure 3.2 shows the path that a Petri net analysis of a SDL system should take to return results applicable to SDL. The SDL system is mapped to a Petri net from which the Markov chain is derived. This is solved and the performance results applied to the Petri net, these are then in turn applied to the original SDL system.

Time in these nets is assumed to be exponentially distributed, so the Markov chain can be derived for analysing the performance. This is not always realistic in specifying real protocols as other distributions may be necessary to represent real-life systems. Instead, the Coxian distribution is introduced, explained next.

3.3.3 Coxian distribution

In many real life cases the simple negative exponential distribution, as mentioned before, does not adequately represent the time distributions. The approach chosen was to thus resort to the Coxian

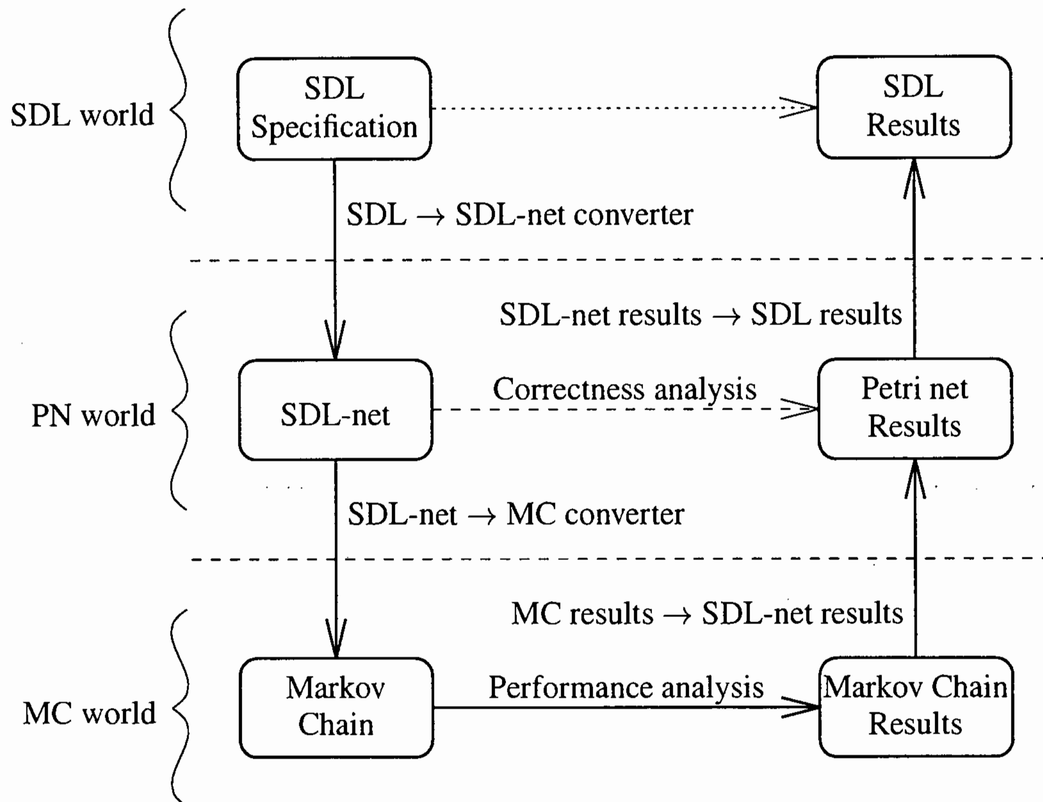

Figure 3.2: Analysis path of SDL using Petri nets.

[Cox55] or phase type [Kan92] distributions.

Distributions related to the negative exponential distribution with coefficient of variation less than 1 are called *hypo*-exponential and with coefficient of variation greater than 1 are called *hyper*-exponential. It will be shown how the Coxian distribution is used to represent these general distributions.

Exponential

Performance evaluation relying on Markov chain analysis requires that time has a negative exponential distribution, illustrated by the single server in Figure 3.3. Let X denote a nonnegative continuous random variable characterising the negative exponential distribution with parameter μ. Then the mean is given by

$$E[X] = \frac{1}{\mu}$$

and the variance by

$$\sigma_X^2 = \frac{1}{\mu^2}.$$

The coefficient of variation CV_X is simply

$$\mathrm{CV}_X = \frac{\sigma}{E[X]} = 1.$$

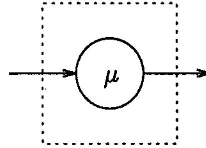

Figure 3.3: Exponential distribution.

Hypo-exponential

An example of a *hypo*-exponential distribution is the n-phase Erlang [SC81, Kan92] distribution, with each phase having exponential distribution with mean $1/\mu$ illustrated in Figure 3.4. For an Erlang distribution X the expression for the mean is given by

$$E[X] = \frac{n}{\mu}$$

and the variance is given by

$$\sigma_X^2 = \frac{n}{\mu^2}$$

which leads to

$$\mathrm{CV}_X = \frac{1}{\sqrt{n}}.$$

For this distribution, $\mathrm{CV}_X \leq 1$ and decreases quite slowly with the number of phases.

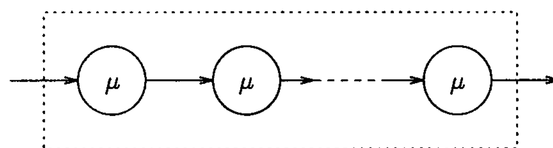

Figure 3.4: Erlang (hypo-exponential) distribution.

Hyper-exponential

An example of a *hyper*-exponential distribution is given by a server with two exponential phases connected in parallel [Kan92] illustrated in Figure 3.5. The two phases have mean service times of $1/\mu_1$ and $1/\mu_2$ respectively, and an arriving customer goes to the first phase with probability α, and to the second phase with probability $1 - \alpha$. For such a distribution, the expression for the mean (or first moment) is given by

$$E[X] = \frac{\alpha}{\mu_1} + \frac{1-\alpha}{\mu_2}$$

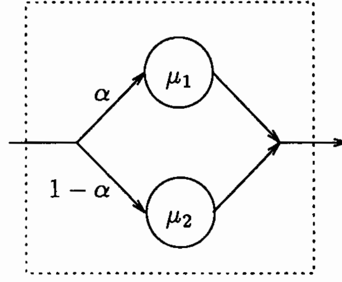

Figure 3.5: Parallel phase (hyper-exponential) distribution.

and the variance is given by

$$\sigma_X^2 = \frac{\alpha(2-\alpha)}{\mu_1^2} + \frac{1-\alpha^2}{\mu_2^2} - \frac{2\alpha(1-\alpha)}{\mu_1\mu_2}$$

which leads to

$$\mathrm{CV}_X = \frac{\sqrt{\mu_1^2(1-\alpha^2) - \mu_1\mu_2 2\alpha(1-\alpha) + \mu_2^2\alpha(2-\alpha)}}{\mu_1(1-\alpha)+\mu_2\alpha}.$$

It is possible to obtain an arbitrarily large value for $\mathrm{CV}_X \geq 1$ by choosing α and μ_1/μ_2 appropriately [Kan92]. $\mathrm{CV}_X = 1$ when either $\mu_1/\mu_2 = 1$, $\alpha = 1$ or $\alpha = 0$. Unlike the Erlang distribution, the value of CV_X for the hyper-exponential distribution does not depend significantly on the number of phases.

Coxian Distribution

The Coxian distribution [Cox55, CS61, Kan92, Ste94, Wal88, BK96, GP87] (or branching Erlang distribution) illustrated in Figure 3.6 is a generalisation of the Erlang distribution that allows a customer to leave the system after completing service at any phase, rather than only after the last phase. The mean rate of each server can also be different. Let X denote a random variable with Coxian distribution, X_i the duration at the ith phase and $E[X_i] = 1/\mu_i$. The probability of going to phase $i+1$ after the customer has been in phase i is represented by α_i, and the probability of leaving the server after phase i by $1-\alpha_i$. From [Kan92] we have

$$\delta_i = (1-\alpha_i)\prod_{j=1}^{i-1}(\alpha_j)$$

The expression for the mean (or first moment) is given by

$$E[X] = \sum_{i=1}^{n}\delta_i\sum_{j=1}^{i}\frac{1}{\mu_j}$$

the second moment is given by

$$E[X^2] = \sum_{i=1}^{n}\delta_i\left(\sum_{j=1}^{i}\frac{1}{\mu_j^2} + \left(\sum_{j=1}^{i}\frac{1}{\mu_j}\right)^2\right)$$

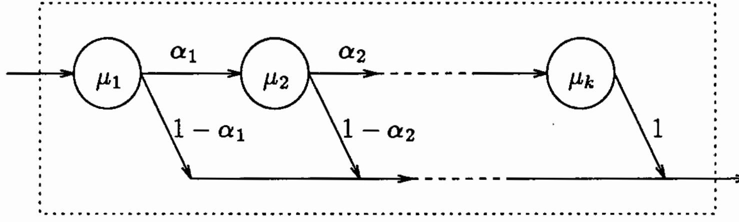

Figure 3.6: Coxian phase distribution.

and the variance is given by

$$\sigma_X^2 = E[X^2] - (E[X])^2.$$

With this distribution, any $CV_X \geq 1/\sqrt{n}$ can be achieved. If one only wants to match a given mean and coefficient of variation, one can impose certain restrictions on the Coxian distribution. It is impractical to expect a general distribution to be approximated by a Coxian phase type distribution with different α_i and μ_i where $i = 1, \ldots, n$ and n the number of phases. In order to surmount this problem, it will be shown how the parameters for the Coxian distribution are calculated given the mean and coefficient of variation for the general distribution.

Representing the Exponential Distribution with a Coxian Distribution

For the case where $CV_X = 1$, the exponential distribution is represented as a Coxian type server with 1 phase and $\mu_1 = 1/E[X]$.

Representing the *Hypo*-exponential Distribution with a Coxian Distribution

For the case where $1/\sqrt{n} \leq CV_X < 1$ it is enough to let $\mu_1 = \mu_2 = \cdots = \mu_n$ and to only let $1 - \alpha_1$ and $1 - \alpha_n$ be nonzero [Kan92, SC81]. Let n be the smallest integer $\geq 1/CV_X^2$. Then α_1 is uniquely determined by

$$\alpha_1 = 1 - \frac{2nCV_X^2 + n - 1 - \sqrt{(n^2 + 4 - 4nCV_X^2)}}{2(CV_X^2 + 1)(n-1)}$$

and

$$\mu_1 = \mu_2 = \cdots = \mu_n = \frac{n - (1-\alpha_1)(n-1)}{E[X]}.$$

If $1/CV_X^2$ is an integer then $\alpha_1 = 1$ and $\mu_1 = \mu_2 = \cdots = \mu_n = n/E[X]$, corresponding to the Erlang distribution.

Representing the *Hypo*-exponential Distribution with a Coxian Distribution

For the case where $\mathrm{CV}_X > 1$ [Kan92, SC81] one only needs n to be at least 2. With $n = 2$ one has three free parameters α_1, μ_1 and μ_2. Another common constraint is that the subservers make an equal contribution to the mean. This leads to

$$\alpha_1 = 1 - \mathrm{CV}_X^2 \left(1 - \sqrt{1 - \frac{2}{1 + \mathrm{CV}_X^2}}\right)$$

$$\mu_1 = \frac{1 + \sqrt{1 - \frac{2}{1+\mathrm{CV}_X^2}}}{E[X]}$$

$$\mu_2 = \frac{1 - \sqrt{1 - \frac{2}{1+\mathrm{CV}_X^2}}}{E[X]}$$

Application of Coxian Distribution in SDL-net

Some service times in SDL-net are represented by a general Coxian distribution. The cases above show how the various distributions related to the *negative exponential distribution* are modelled with the Coxian distribution. The formulae for the various cases may be used to calculate the parameters for the Coxian distribution, given the *mean* and the *coefficient of variance*.

This page intentionally left blank.

Chapter 4

Mapping of SDL to SDL-net

In this chapter some general aspects of mapping SDL to the SDL-net are discussed, three SDL constructs are examined which are particularly difficult to map to a net and three new places for the mapping are defined. It is also shown how some other SDL constructs are mapped to the SDL-net.

4.1 Mapping SDL in general

4.1.1 Dynamic semantics

The dynamics of a SDL system [IT93a] are interpreted by a number of concurrent meta-processes as illustrated in Figure 4.1. The *system*-process is the link to the "outside world". It is created first and then creates one *view*-process instance, one *timer*-process instance, one *path*-instance for each delaying path by which SDL signals may be transported and one *process-set-admin*-process instance for each set of process instances in the SDL system.

The *process-set-admin*-process manages several processes as illustrated in Figure 4.2. The *process-set-admin*-process is the "starting-point" for interpretation of a SDL process instance set. It creates one instance of the *input-port*- and *sdl-process*-processes for every new SDL process instance created. If the SDL process is represented by services, the *sdl-process*-process creates one *sdl-service*-process for each service.

SDL-net Mapping

These individual meta-processes are mapped to SDL-net sub-nets which will be discussed in greater detail in this chapter.

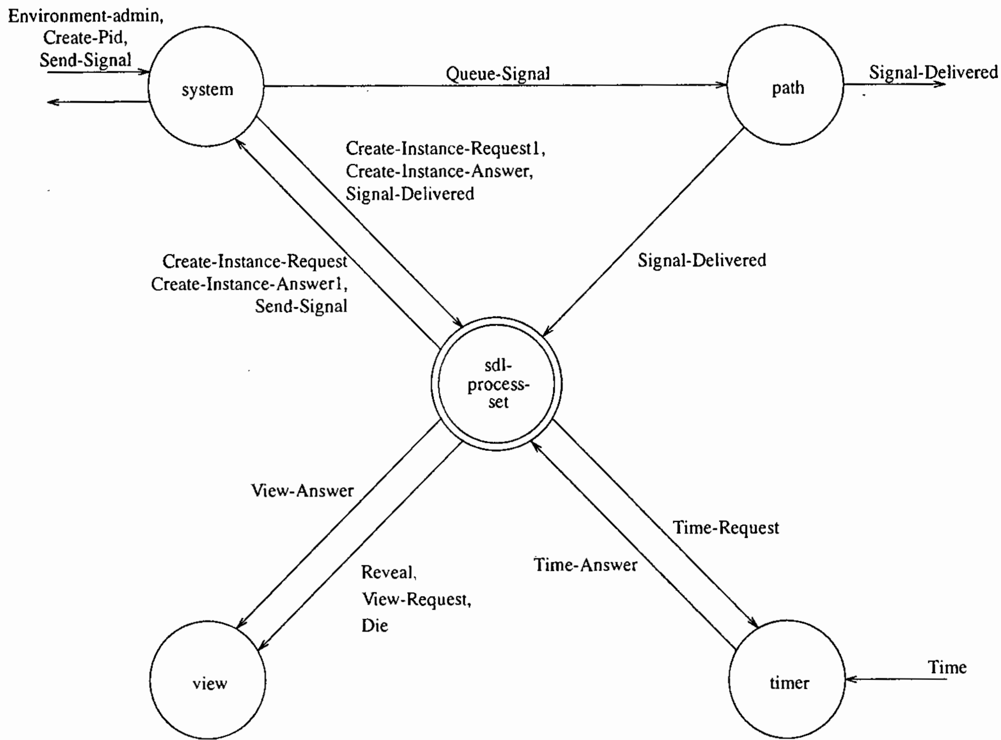

Figure 4.1: Overall structure of Dynamic Semantics Interpretation Model

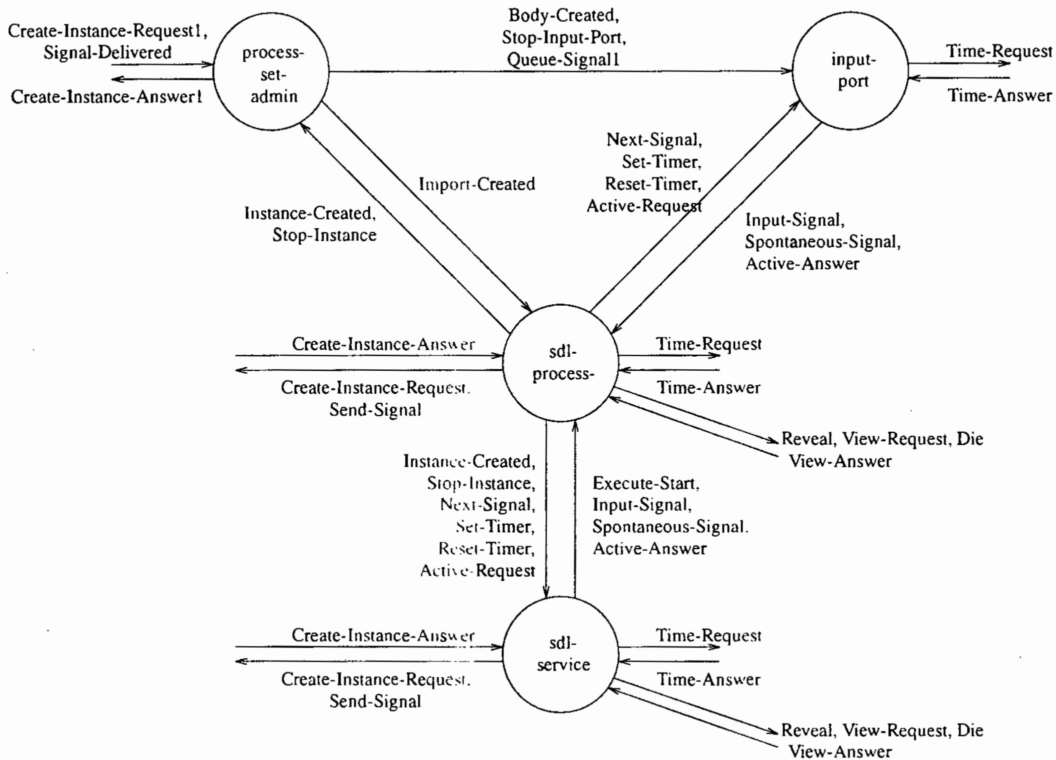

Figure 4.2: Structure of Dynamic Semantics Interpretation Model for SDL Process Instance Set

4.1.2 Extended Finite State Machines

A process defined as a *state machine* must contain a finite number of states and may only be allowed to be in one state at a time. It moves between states on receiving some external stimuli, such as an arriving signal, or by an internal stimuli, such as a spontaneous transition. When no signals are present in the signal queue and a continuous signal condition is enabled then the continuous signal may also act as an internal stimuli.

There is a fundamental difference between the global state of the system and process states. The global state includes information of what signals are in which queues, the state of each process, the value of each data variable, etc. Process states, also known as the *semantic states* of the process, are the states in the EFSM used to define a process.

The state machine used in the SDL processes is an extension of the deterministic Finite State Machine (FSM). The FSM has a finite internal state memory and operates with a discrete and finite set of *inputs* and *outputs*. For each combination of *input* and *state*, the memory defines an *output* and the *next state*. A limitation of the FSM is that all information which needs to be stored must be represented in explicit states. Although it is possible to represent most systems in this way, it may not always be practical. There may be values to store which are significant for the future behaviour but do not contribute greatly to the overall understanding of the system. This information should not be a part of the explicit state space as it will clutter the presentation. For such applications, the FSM may be extended with auxiliary data. Signal addressing information and sequence numbers are examples of auxiliary data.

The SDL Recommendations define two auxiliary operations on the auxiliary data, which may be included in transitions of the Extended Finite State Machine (EFSM), namely *decisions* and *tasks*. *Decisions* inspect the parameters associated with inputs and information in auxiliary data when such information is important for the sequencing of the machine. *Tasks* perform functions on auxiliary data.

The interactions between the machines are represented by signals, i.e., the EFSM's receive signals as input and generate signals as output. SDL allows for the possibility of non-zero transition time, and defines a FIFO queueing mechanism for signals which arrive at a machine while it is executing a transition. Signals are considered one at a time, in order of their arrival, except when the *save* construct is used as described in Section 4.2.1 on page 27.

Transitions between states are usually regarded as taking zero time. Since this is not realistic in modelling the performance of systems, in this thesis the transitions may be assigned exponential rates. These are assigned automatically depending on the actions that are performed during the transition.

SDL-net Mapping

Each state in a SDL process is mapped to an ordinary SDL-net place with the number of tokens representing the number of process instances in that state.

It may sometimes be convenient to specify that a transition should occur for more than one process state. This is only a shorthand notation which needs to be expanded before mapping the process to a SDL-net. As an example of such a shorthand notation, when a transition applies to states **A** *and* **B** the state descriptor for that transition is specified as **A,B**. When a transition applies to every process state the state descriptor may be described as simply *. If a transition applies to every process state *except* some state **A** then this is specified by ***(A)** in the state descriptor.

In Figure 4.3 the process states are **State A**, **State B** and **State C**. The state descriptors are defined in the graphical SDL state symbols and they are **State A**, **State B,C**, **State *(B)** and **State ***. The states in the state descriptor **State B,C** could also be represented separately as **State B** and **State C** each having an input transition with signal **s1**. Similarly, **State *(B)** could be rewritten as **State A** and **State C**, i.e., all process states *except* **State B**. Both states would have an input transition for signal **s2**. **State *** could be rewritten as **State A**, **State B** and **State C**, i.e., all process states. All three states would have an input transition for signal **s3**.

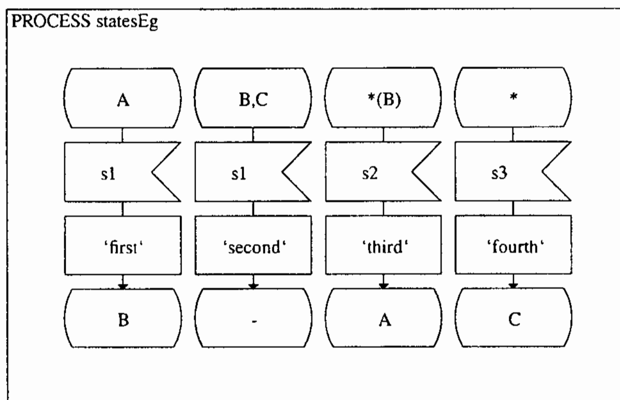

Figure 4.3: Example of valid process states

When a SDL process is mapped to a SDL-net the state descriptors first are rewritten to the equivalent process states and the respective transitions copied. Figure 4.4 shows the expanded representation of Figure 4.3 so that **State A** includes the input transitions for signals **s1**, **s2** and **s3**. **State B** includes the input transitions for signals **s1** and **s3**. **State C** contains input transitions for signals **s1**, **s2** and **s3**.

The SDL process shown in Figure 4.4 is represented as the SDL-net in Figure 4.5. Each semantic SDL state is mapped to an SDL-net place, so **State A** is mapped to a place **STATE!a**. Each input is mapped to a SDL-net input transition, so **Input s1** at **State B** is mapped to a transition **b+s1**.

As signals arrive at the process, they are put in a signal queue on an input place, also called a Save-place. This is discussed further in Section 4.2.1. Usually there would be arcs from this input place to the various input transitions to indicate a signal being consumed. These "input arcs" were omitted to simplify the diagram. When a token is in the place **STATE!a** then, depending on the signal in the input queue, the token is removed from the place **STATE!a** and put in the next state. For instance, if a token is in **STATE!a** and transition **a+s1** fires, then a token is put on **STATE!b**; this corresponds to the SDL process changing its state from **State A** to **State B** upon receiving signal **s1**.

As will be seen in the next chapter, a signal is discarded when no explicit input transition is defined for that signal. This is called an "implicit input". An example is if signal **s2** had to arrive whilst the process above was in **State B**. This is specifically shown by the SDL-net transition **b+s2**.

4.2 Special constructs

Some of the constructs in SDL that are difficult to represent in Petri nets are shown in this section, and three new places are introduced that are used specifically to model these constructs.

4.2.1 SAVE Construct

The process signal queue is represented by the meta-process *input-port* as described in Section 4.1.1 on page 23. Signals are stored in the *input-port* as they arrive at the process, and in SDL there is no limit on the number of signals that may be in that queue.

At each process state a number of possible signal inputs are defined. These are also called *explicit* inputs. Signals arriving at the process are usually consumed by these explicit inputs in a FIFO order.

However, if at some state a signal arrives for which no input was defined, it is simply discarded. These "inputs" are also called *implicit* inputs and are usually expanded to inputs which contain no actions in their transition body and which stay in the same process state.

Sometimes a signal may arrive at a process state where it cannot be dealt with immediately. To prevent the signal from being discarded as an implicit input, the *save* construct may be used to keep the signal in its position in the signal queue until the next process state. The next signal in the queue is in that case considered for input.

Several signals may be saved at each state. This is represented by either having one *save* construct for each signal, or by writing all the signal names in the *save* construct. If several signals are to be saved, the semantics of the *save* symbol implies that the order of their arrival is preserved.

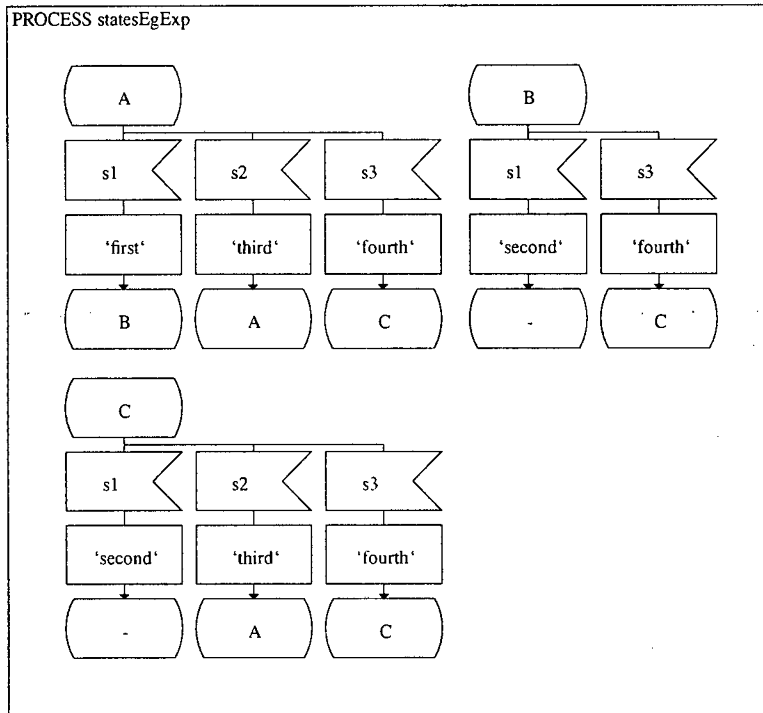

Figure 4.4: Expanded process of Figure 4.3.

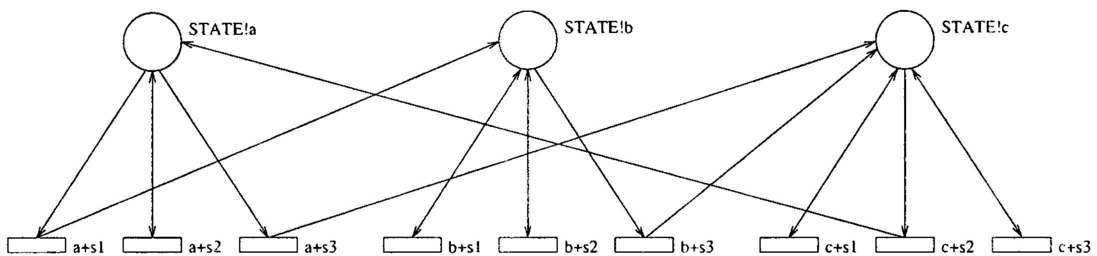

Figure 4.5: SDL-net mapping of Figure 4.4.

An example of a SDL process in which the *save* construct is used is given in Figure 4.6. If the order of arriving signals is s_2, s_1, s_1, s_1 with s_2 being first to arrive, the transitions that occur are: A $\xrightarrow{s_1}$ B, B $\xrightarrow{s_2}$ A, A $\xrightarrow{s_1}$ B, B $\xrightarrow{s_1}$ B. Note that B $\xrightarrow{s_1}$ B represents an *implicit* input, also called a *discard*. Figure 4.7 illustrates the signal queue at each state, the signal that is consumed, and the next state following it.

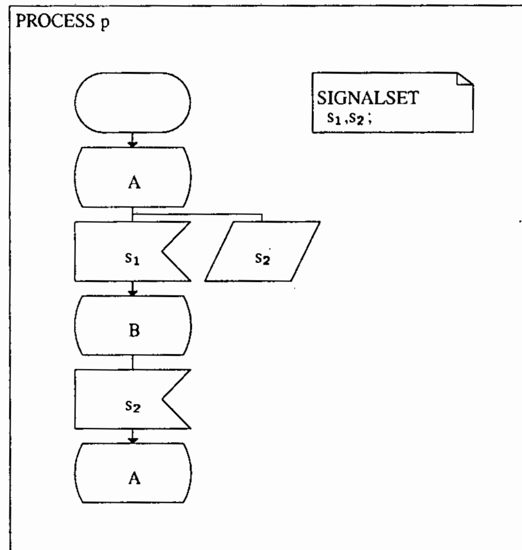

Figure 4.6: SDL process using the save construct.

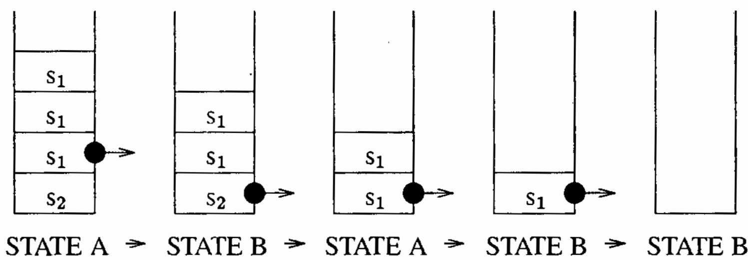

Figure 4.7: State and Signal queue changes with Save construct.

The Save-place

To represent this complex scheduling in Petri nets, a new place is introduced, which is called the Save-place and is based on the notion of a "special" place [Bau93]. This is formally defined in Section 5.2.1 on page 58.

The Save-place consists of a *FIFO queue*, a variable *active* storing the last available signal, and a *depository* [Bau93], as illustrated in Figure 4.8. The initial state of the Save-place is the *FIFO queue* empty, the *depository* empty and *active* set to "ϵ".

Each SDL signal that can arrive at a process is represented as a coloured token. When such a signal token arrives at a Save-place, it is inserted at the tail of the *FIFO queue*. Once an SDL

signal reaches the head of the signal queue it becomes available as an input which is represented in the Save-place by a **copy** of the signal token at the head of the queue being placed in the *depository*. In addition, to enable complex scheduling with save, the *active* variable is set to the colour of the available signal token. The token is then available to the SDL-net transitions. Once it is consumed from the *depository*, it is removed from the *FIFO queue* and the *active* variable is set to "ϵ". This is necessary to preserve the order of the tokens in the queue, especially when signals are saved.

When a SDL signal must be saved at a process state, the Save-place must be instructed to not release that signal to any transition. This is done in SDL-net by introducing a special *control token* for each signal that has to be saved. When the SDL process enters a state where a signal must be saved, then the control token of that signal is sent to the Save-place which prevents the saved signal from being released from the *FIFO queue*. These control tokens do not enter the *FIFO queue*, but are rather transferred to the depository directly. They can be removed from the Save-place immediately if the process enters a new state where those signals should not be saved.

SDL-net Mapping

For each unique signal that can arrive at the input port, a separate colour is created in the colourset of the Save-place. The valid set of unique input signals is the union of the sets of

- signals in all signal routes leading to the process.
- signals that the process sends to itself, specified in the *signalset* definition.
- timer signals.

Usually, only a small subset of signals are saved in a process. For each signal that is saved, a control token is added to the colourset of the Save-place.

For each explicit input, a timed SDL-net transition is created. If the delay is zero, then an immediate transition is created instead.

If a signal is not covered by an explicit *input* or by a *save*, then an implicit input transition is created so the process consumes the signal but stays in the same process state.

SDL-92 introduced the *spontaneous transition*, a transition that can happen without an external stimuli; this is represented by a transition that does not consume a signal from the Save-place.

When a transition leads to a SDL state where some signal is to be saved, a control signal token is sent to the Save-place. While the SDL process is in that state, the saved signal does not become

available for input. Each transition leading from the state has to remove the control signal token from the Save-place.

The Save-place only needs to support the input and the saving of signals. If enabling conditions or continuous signals are specified, transformations [IT93b] may be applied to represent the same system using only input and save.

For the process in Figure 4.9, the valid input set is **s1**, **s2** and **s3**. **State A** has an explicit input transition for **s1** and a save transition for **s3**, but no transition is specified for **s2**. The process including the implicit transition for **s2** is shown in Figure 4.10.

The expanded SDL specification shown in Figure 4.10 is mapped to the SDL-net shown in Figure 4.11.

4.2.2 TIMER

SDL processes can contain timers that expire at a set time. A timer is considered inactive until it is set for the first time. It is then active until it either expires and is consumed, or until it is reset. If an *active timer* is set again, it remains active, but a new expiry time is set. Similarly, when an inactive timer is reset, it remains inactive. When the timer expires, a timer signal is put into the back of the process signal queue. An active timer can be reset in which case it is stopped and its timer signal removed from the signal queue if it has already expired.

SDL-92 has a shorthand form for the timer in which a timeout value is assigned to the timer with a *duration ground expression* which is used to compute the expiry time. For example, timer **T** could be defined in SDL-92 as **timer T:=5.0** which means that the standard timeout will be in 5 time units from when it is set. The timer is set with **set(T)** which in the SDL-88 equivalent is **set(now+5.0,T)**.

The Timer-place

The Timer-place of SDL-net was developed to account for this semantic behaviour. A Timer-place of a SDL-net is a special place which models the deterministic timing delay of the *timer* construct with a Coxian service time distribution. There is one Timer-place for each timer in the system, so only one server is needed for each Timer-place. The timer may be reset, in which case it needs to be stopped regardless of which Coxian phase it is in. To stop the timer, a special *reset token* is introduced that interrupts the server.

The Timer-place consists of: a *variable* marking the current Coxian phase of the timer, and a *depository* for the expired timer tokens and *reset tokens*, as illustrated in Figure 4.12.

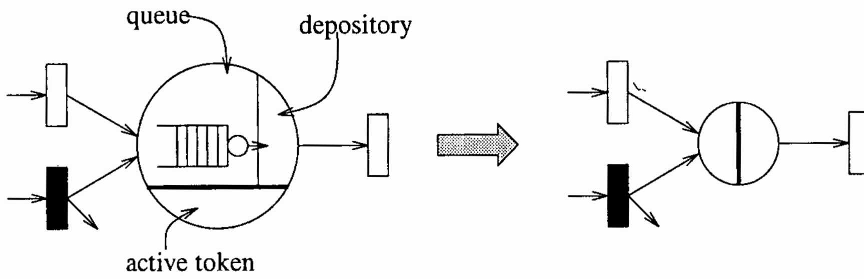

Figure 4.8: A Save-place and its shorthand notation

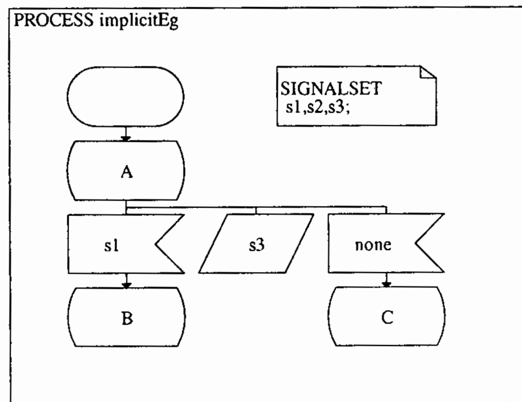

Figure 4.9: SDL Process where implicit input is not expanded.

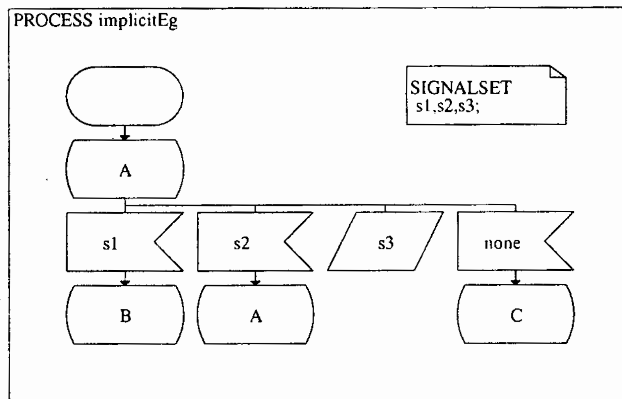

Figure 4.10: SDL Process with implicit input for signal **s2** expanded.

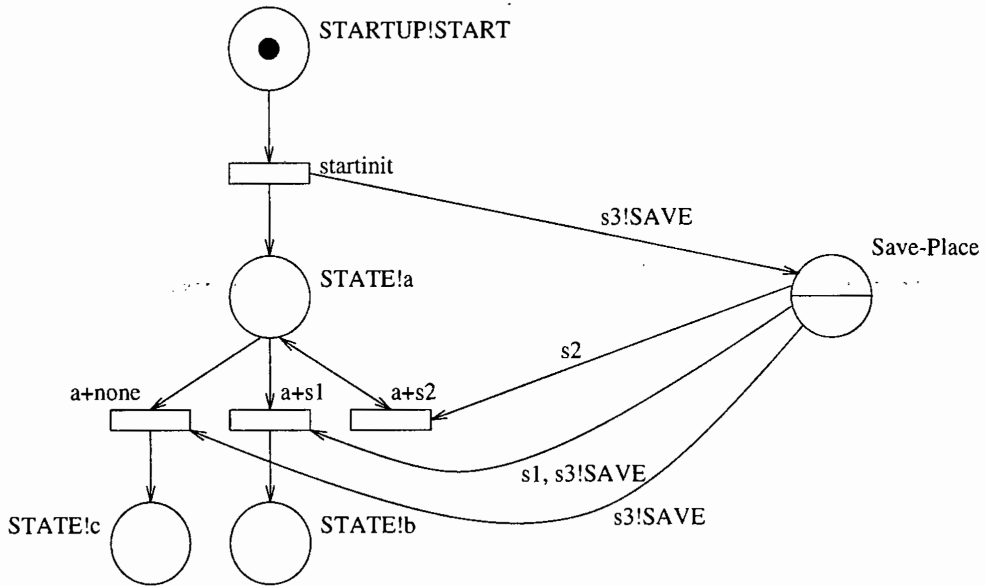

Figure 4.11: SDL-net mapping of Figure 4.10.

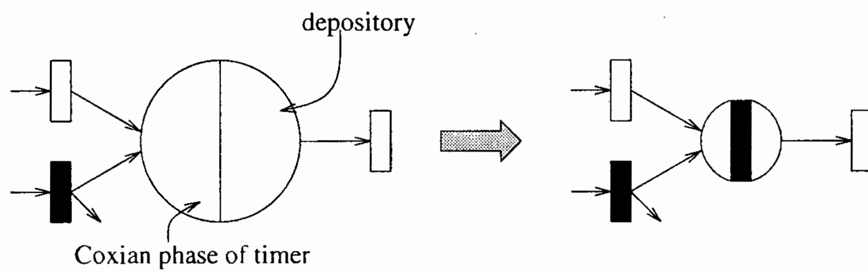

Figure 4.12: A Timer-place and its shorthand notation

A token that arrives at a Timer-place either represents a SDL timer *set* or a SDL timer *reset*. If a set token arrives and the timer is inactive, then the service is started so the token enters the first phase of the Coxian service time distribution. If the timer is already active, the token is put in the depository without disturbing the active timer. This situation cannot occur in SDL, because an active timer is always reset before it is set again and is mentioned for completeness only.

If a reset token arrives, it is put directly into the depository. If the timer is active, then the service is stopped and the set token is put into the depository.

SDL-net Mapping

For a realistic mapping of the timer, a close approximation to the *deterministic timeout* needs to be specified. In addition, supporting net constructs need to be defined which allow for the *set*ting and *reset*ting of timers. In this translation, only one timeout value may be declared for each timer. This corresponds to the SDL-92 definition of a timer with a *Duration ground expression*.

To approximate the *deterministic timer*, a Coxian distribution is used as discussed in Section 3.3.3 on page 16. The user can specify the number of phases in the Coxian distribution together with the rate of each phase and the probability of exit. The exact parameters are determined given the mean and coefficient of variation by applying the formulae in Section 3.3.3.

A net is created to control the setting and resetting of each *timer*. As this net is quite complicated, the illustration is split into 4 diagrams. Transition and place labels in the diagrams that are the same, refer to the same object. The places labelled **active** and **inactive** represent a condition for the state of the timer.

Figure 4.13 - "Reset" net-construct for timer t If the timer is inactive when it is reset (i.e. a token is in place **t!inactive**), then **t1** fires, and the timer remains inactive. If it is active (i.e. a token is in place **t!active**), then **t2** fires, placing a token in **t!reset!deactivate** and a reset signal **#t** is sent to the Timer-place. An active timer can either still be running, or it could have expired and sent a timer signal to the signal queue. If it is still running, then **t4** fires and timer token **t** is removed from the depository of the Timer-place, together with the reset signal. If it has expired already, then **t3** fires and timer token **t** is removed from the Save-place, together with the reset signal from the Timer-place. If a timer expires, **t8** can fire so the timer joins the Save-place.

Figure 4.14 - "Save" net-construct for timer t When a timer is reset and its timer signal needs to be removed from the signal queue, it is removed irrespective of its position in the queue. This could create a problem in the Save-place if only the signal at the head can be removed. The timer signal can however be forced to the head of the queue, by saving all other possible

signals in the Save-place. Only the timer signal is then available, and can thus be consumed. The net in Figure 4.14 illustrates the case where the valid input set of signals to the process consists of signals **s1**, **s2** and timer **t**. Signals **s1** and **s2** are saved by sending control signal tokens @**s1** and @**s2** to the Save-place. These signals are then again removed at transitions **t3** and **t4**, and the previous order of signals in the Save-place is restored.

Figure 4.15 - "Set" net-construct for timer t If the timer is inactive when it is set, then **t6** fires, and the timer set token is sent to the Timer-place. If it is active already, then **t5** fires, and the *timer* is reset using the *reset* net construct described above. Once the timer has been reset, **t7** fires, and the control goes back to the **set!begin** place. This will enable **t6**, because the timer is now inactive, and the transition **t6** will complete as described before.

Figure 4.16 - Additional edges for timer signal input An active timer becomes inactive when it is either consumed at the Save-place, or when it is reset. It is necessary to add additional edges when a *timer* signal is consumed implicitly or explicitly so that it becomes inactive. In the diagram, the transition **a+t** represents an explicit input of **t** when the process is in **State a**. Transition **b+t** represents an implicit input of **t** when the process is in state **b**. In both cases, the timer is set from active to inactive.

4.2.3 CHANNEL

The EFSMs in SDL communicate by sending signals along signal routes and channels. Channels are used specifically for inter-block communication and signal routes are used for inter-process communication.

The standard queueing strategy for signals moving along channels is First-In-First-Out. In SDL one may define a substructure for a channel which is similar to block substructures, which allows the channel to be defined in more detail. As a standard channel preserves the FIFO order of the signals, it is not possible for signals to overtake each other on one channel. However, it is possible that the order of two signals is changed when they are sent by one process along different channels.

An output signal needs a destination process for the system to be valid. In SDL-92, if more than one destination process exists, then a random decision is taken of where the signal is delivered, while in SDL-88, such a situation is considered a *run-time* error.

Each output signal is sent to specific *process instances* by specifying a *process identifier* (*PId*) as the *TO* value. A path from the *sender process* to the process specified by the *PId* has to exist, otherwise, a *run-time* error will occur. An output signal may also be sent via a specific signal route in which case the *VIA* construct is used together with the name of the *signal route*. SDL-92 has expanded the *VIA* construct to allow a signal to be broadcast to all possible processes with the *VIA ALL* construct.

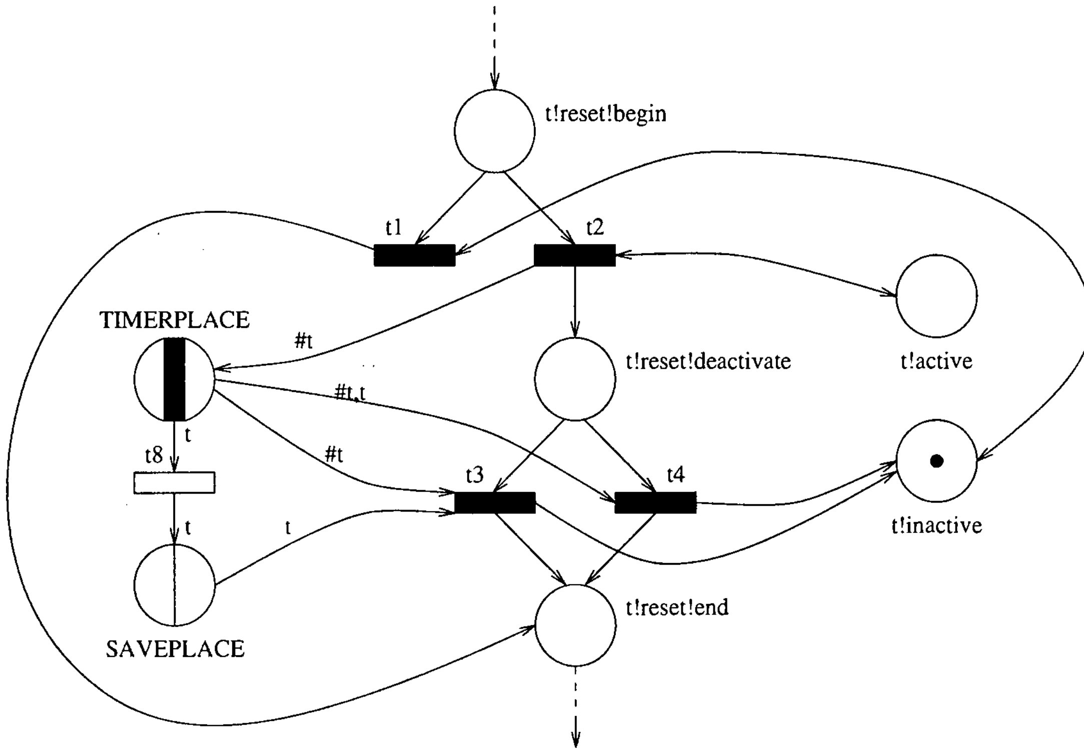

Figure 4.13: "Reset" net-construct for timer t.

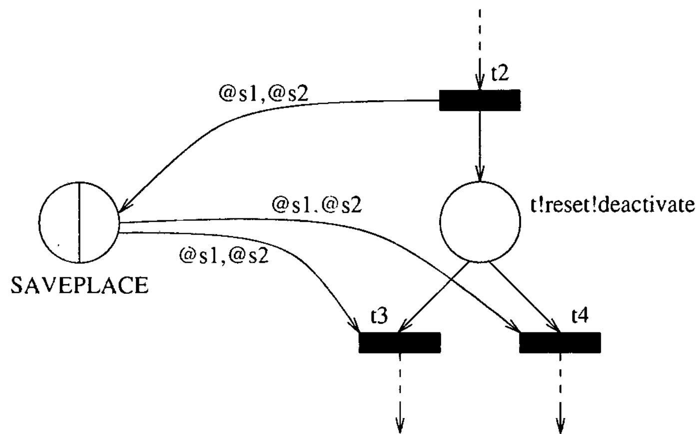

Figure 4.14: "Save" net-construct for timer t.

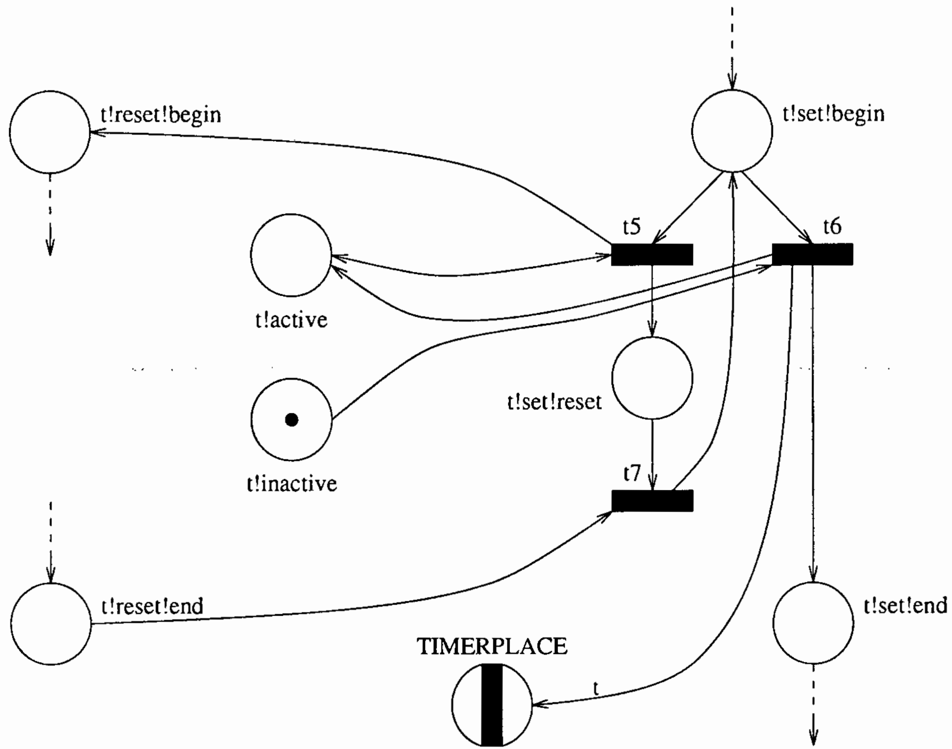

Figure 4.15: "Set" net-construct for timer t.

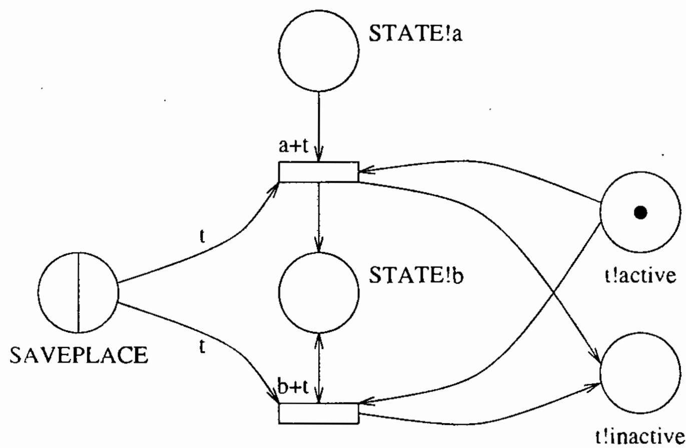

Figure 4.16: Additional edges for timer signal input.

A signal may be defined at various hierarchical levels of the system and it consists of a name and a description that defines the data it may contain. It may traverse several channels and signal routes while travelling from one process to another.

The most important factor for analysing the *performance* of a channel are the delays associated with the transfer of signals; these can have a very strong influence on the performance of a system. In SDL, the delay is represented by a *timing delay function* (*delayf*) associated with the channel. This returns a *boolean* value indicating whether the time delay has expired or not. The time distribution for the *channel delay* is not specified, so an exponential time distribution can, for example, be used to model the function *delayf*.

In SDL-92 a channel is set to transfer signals with no delay by following the channel description with the keyword *nodelay*. The graphical description of the channel is in that case the same as that of the signal route. This is illustrated in Figure 4.17 with timed channel **data** and nodelay (immediate) channel **control**. Communication along a signal route is considered to have no transmission delay.

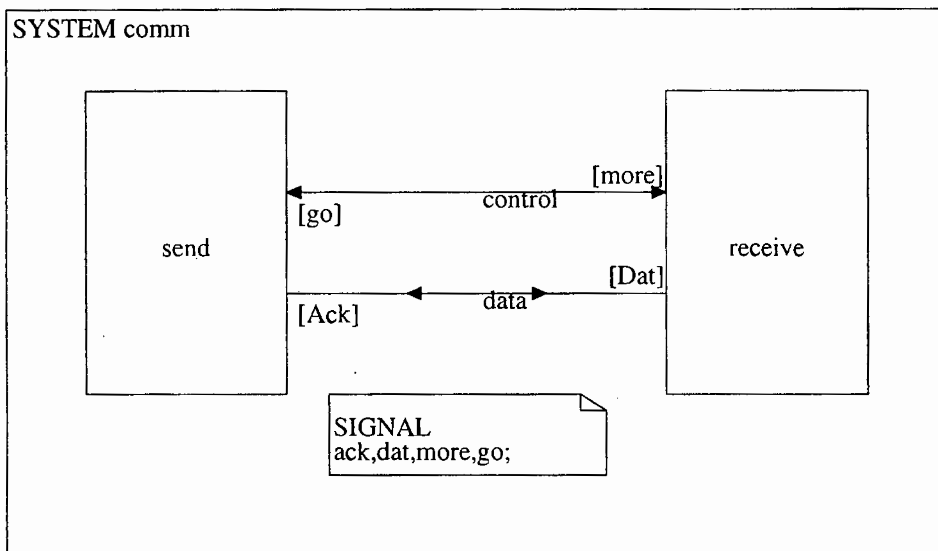

Figure 4.17: System with both immediate and delayed channel

The Channel-place

A Channel-place of a SDL-net, is a special place which models the FIFO channel semantics with Coxian service time distribution delays.

The Channel-place consists of a *queue* for all incoming tokens representing signals, an variable *active* representing the signal in service together with its current Coxian phase and a *depository* for the signal tokens that have completed service. This is illustrated in Figure 4.18.

When a token representing a SDL signal arrives, it joins the back of the queue. The tokens are served in FIFO order and the delay is modelled by a Coxian distribution. Only one token may be served at a time. If more than one token was allowed to be served, then, because it is a Coxian distribution, tokens would be able to overtake each other, thereby invalidating the FIFO order of the channel. Once a token has completed service, it is put in the depository. There is no blocking of tokens in the depository, because this would impose an artificial delay on the tokens.

In a Concurrent Communicating System, signals may have different transfer times due to, for instance, varying packet sizes. In the SDL-net, however, a signal independent service time for each channel in the system is assumed. Thus each signal that is transferred across a certain channel has the same delay. The SDL-net does not inherently prevent the use of signal dependent service times, however, this would complicate the specification of the model significantly and is therefore not used.

SDL-net Mapping

Signals are sent along ordinary channels according to a timing delay specified by the user and which may be a general time distribution delay approximated by a Coxian service time distribution. A Channel-place is created for each direction of an ordinary channel. Immediate transitions are created to put signals onto the Channel-place and to remove them from the Channel-place.

For each signal on a *signal route* and *nodelay channel* an immediate transition is created that routes the signal.

A *connect* is used as an interaction point between channels and signal routes on different levels of the SDL system. In SDL-net, the *connect* is represented by a place containing the colourset of signals consisting of the union of all signals arriving at and leaving from the *connect*. The signals are routed to and from the *connect* place with immediate SDL-net transitions.

The SDL system shown in Figure 4.17 has two channels, channel **control** being immediate and channel **data** being delayed. The corresponding SDL-net is shown in Figure 4.19 and includes a Channel-place for each direction of the **data** channel and immediate transitions for the routing of signals along the *immediate* channel.

4.3 Other SDL constructs

In this section the structure of SDL systems is dealt with, and it is shown how process create and process stop are mapped to the SDL-net. It is also shown how process outputs, process data, and process decisions are mapped to the SDL-net.

4.3.1 Structure of SDL Systems

SDL allows the various hierarchical constructs to make system design more manageable.

The *system* is the highest level of hierarchy in SDL. Each system may contain blocks which may be connected by channels. An *open* system contains at least one channel that connects a block to the environment. A *closed* system contains only channels connecting blocks to each other.

Each *block* may contain either processes or sub-blocks but SDL does not allow the mixing of processes and blocks on the same level of a hierarchy except when block substructures are used. Processes communicate with each other and the environment by *signal routes*.

Each *process* may either contain a state machine or one or more services. Services are used to further partition the process state machine.

Procedures are used to represent parts of state machines. In SDL-88, *procedures* could only be defined and used locally inside a process, but SDL-92 introduced the remote procedure which is a globally defined procedure.

The *macro definition* construct is used to define parts of the system. A macro is visible globally and can thus be referenced throughout the system.

SDL-92 introduced generic types that are collected in a *package* to make the re-use of SDL designs possible. These types may be instantiated and then used as ordinary structural components.

A typical SDL hierarchy is illustrated in Figure 4.20. At the top level is the system which contains two blocks. The block on the left-hand side contains three processes of which one contains a procedure and one a macro. The block on the right-hand side contains a substructure which in turn contains a block which has a process defined in it.

SDL-net Mapping

The SDL-net is defined as a non-hierarchical Petri net, so all the various levels of a SDL system are "flattened" onto one net. In the mapping, the name of each place and transition has a *prefix* which indicates the type and name of the associated SDL construct and its position in the SDL hierarchy.

To map a SDL system to a SDL-net, all the channels need to be represented as Channel-places as mentioned before, and these need to be connected to transitions that route the signals to the appropriate blocks.

Each SDL block has a connect place for incoming and outgoing signals and immediate transitions representing *signal routes*.

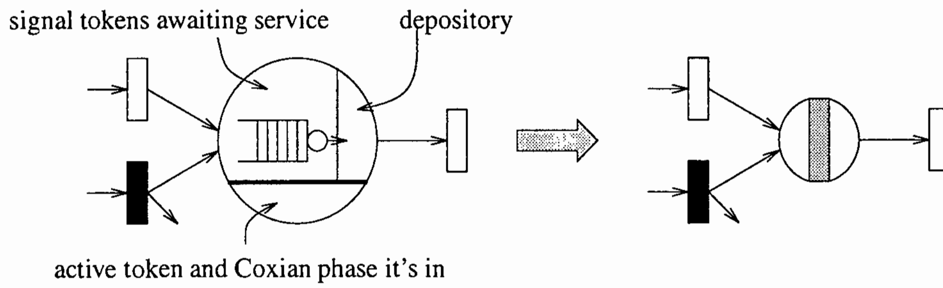

Figure 4.18: A Channel-place and its shorthand notation

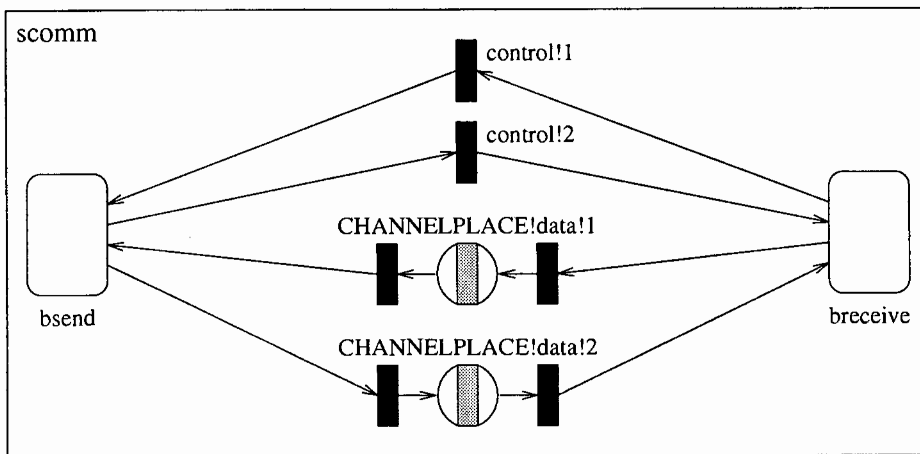

Figure 4.19: Equivalent generated SDL-net of Figure 4.18.

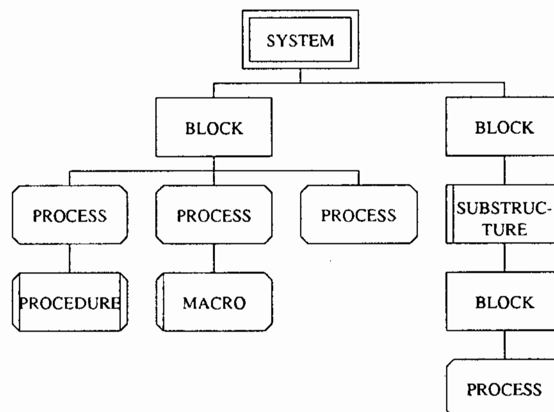

Figure 4.20: SDL hierarchy.

The mapping of SDL processes is covered in the rest of this chapter and includes dynamic process creation and termination, ordering of incoming signals, etc.

Figure 4.21 illustrates the SDL hierarchy of a simple Private Automatic Branch Exchange (PABX) that includes a SDL block containing a switch and a SDL block containing three phones. Such a hierarchy would be mapped onto a SDL-net structure as illustrated in Figure 4.22 which shows the prefixes for the SDL-net constructs at each level. The prefix for the system pabx is simply **main.spabx** and is used for any SDL-net object defined at that level. Similarly, the block switch contained within the system pabx, has as prefix **main.spabx.bswitch**, and the process switch within that block uses prefix **main.spabx.bswitch.pswitch** for all its objects.

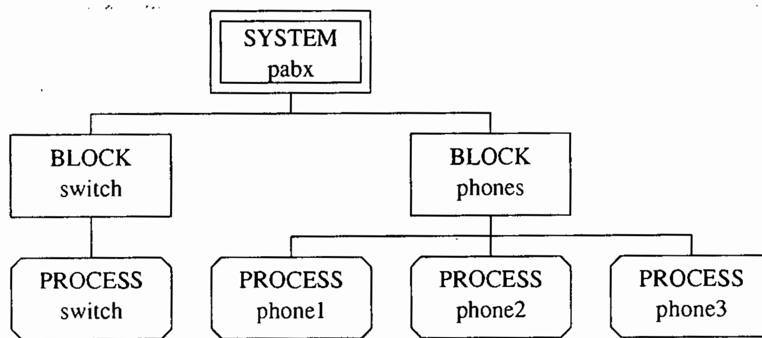

Figure 4.21: Hierarchy of a simple PABX specified in SDL

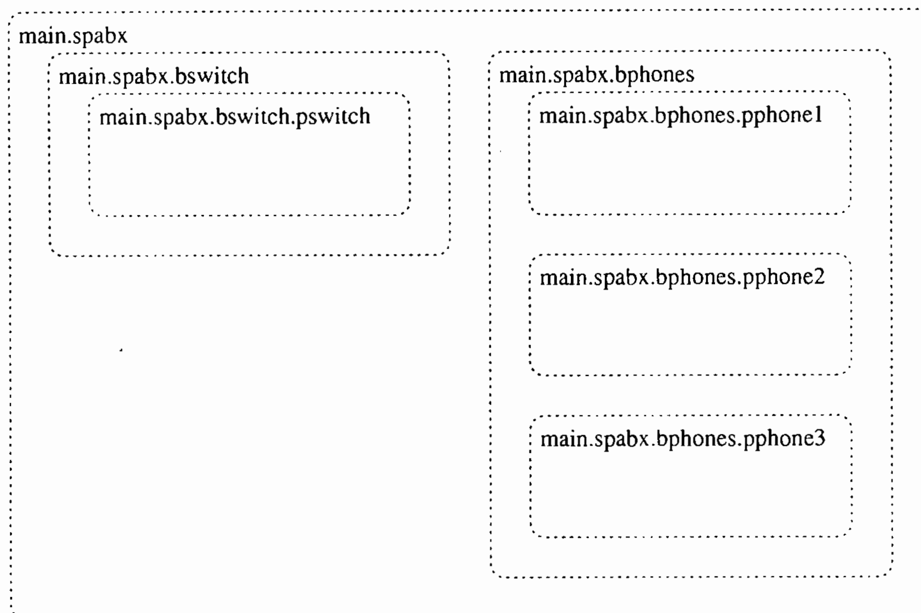

Figure 4.22: SDL-net of PABX structure

The structural concepts mapped to SDL-net are the *system*, *block*, *sub-block*, *process* and *macro definition*. Partial specifications, where the highest hierarchical level is a block or a process, may be analysed as long as they are closed.

4.3.2 Process Create and Process Stop

Processes can have an initial number of instances and a maximum number of instances. If no initial and maximum instances are specified then these default to 1 and infinite respectively. In practice it is not realistic to have an infinite number of processes, as there will always be an upper bound on every machine for the maximum number of active processes. If the maximum number of instances is infinite, then in theory, a process can keep on creating new processes. In practice this may lead to an unbounded number of processes being created resulting in a dynamic error in the SDL specification.

If a *process create* is requested, and the maximum number of processes is already active, then the request is ignored. The process variable *offspring* is set to *null* to indicate that the *create request* was not successful.

A process can only create instances of other processes if they are defined within the same block. However, if a process in another block needs to be created, then a "setup" process could be designed in that block which can cause a process to be created upon receiving a special signal.

A process instance is stopped if it encounters the *stop* construct as a state transition terminator. All process instances have the same priority so it is not possible for a process instance to *kill* another process instance, not even in a parent/child relationship.

SDL-net Mapping

A net is created for each process to manage the creation and termination of process instances. If a maximum number of instances is specified, then this net has to ensure that no more instances are created than the maximum. It also has to free resources when a process is stopped.

The SDL to Place/Transition net mapping in [Gra90] defined a new net for each process instance with one token representing the current state of the process. A disadvantage of creating a new net for each process instance is that the maximum number of instances needs to be known in advance. Even though there will always be a physical limit imposed for the maximum number of process instances, it is not always possible to specify the maximum number of instances in advance. In addition, the size of the net structure grows substantially as each process instance is duplicated as a net.

In the SDL-net mapping, only one net is created for a process. Each new instance of a process is represented by a new token to indicate which state that process is in. In this mapping, the maxmimum number of processes does not need to be known in advance. The disadvantage with this mapping is that it is difficult to determine the probability of a particular process instance being

in a certain SDL state because only the number of instances are shown in a state and not which process instances they are.

Processes may be specified as: **PROCESS send(Initial,Maximum)** where **Initial** is the initial number of instances of the process, and **Maximum** is the maximum number of instances of the process. When the maximum number of instances is specified, the SDL-net mapping has to ensure that the number of instances does not exceed this. If the maximum number of instances are already active, and another token is put in the place **STARTUP!CREATE**, then it is removed with the immediate transition **limit**. The number of active instances is represented by the place **STARTUP!ACTIVE**.

This is illustrated by the SDL/GR representations of processes in Figure 4.23, and the subsequent SDL-net mappings.

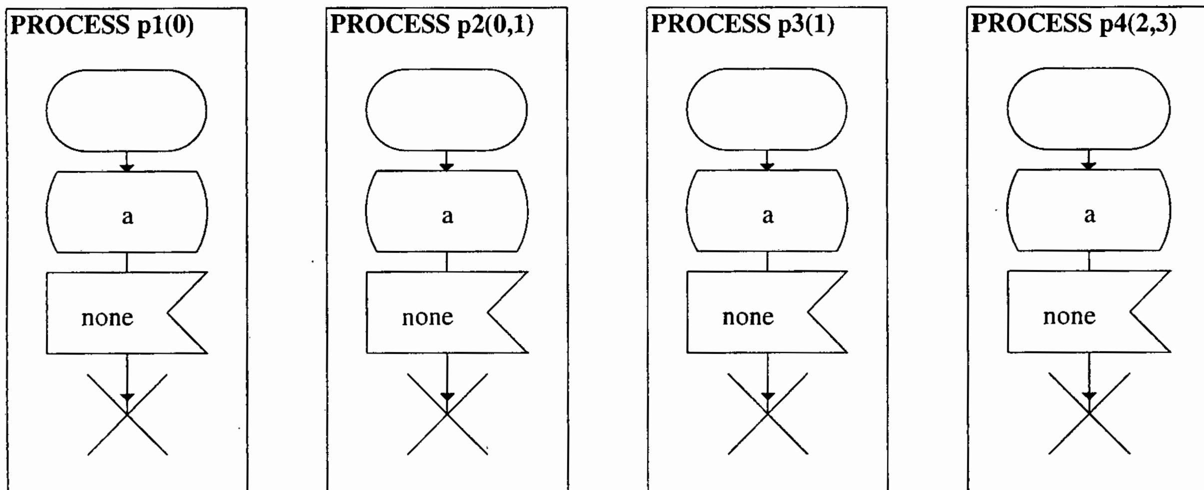

Figure 4.23: SDL Processes with different initial and maximum instances

Figure 4.24 shows the mapping to SDL-nets of processes p1 and p2 defined in Figure 4.23 which both have zero initial instances so that the initial marking of place **STARTUP!CREATE** is 0.

Process p1 has no limit on the maximum number of instances, so the **create** transition is enabled whenever a token is in place **STARTUP!CREATE**. When the process p1 is stopped, it removes the token from place **STATE!a** and does not go to another SDL state.

Process p2 has a limit of 1 on the maximum number of instances, so the **create** transition is only enabled when a token is in both place **STARTUP!CREATE** and place **STARTUP!MAXINST**. When the process p2 is stopped, it removes the token from place **STATE!a** and adds it to place **STARTUP!STOP**. The process instance can then be re-used to create another instance. The **limit** transition ensures that any request for creating more than the maximum number of instances is discarded.

Figure 4.25 shows the mapping to SDL-nets of processes p3 and p4 defined in Figure 4.23 which

both have some initial instances. The initial marking of place **STARTUP!CREATE** is therefore non-zero. The maximum number of instances of p3 and p4 are infinite and 3 respectively.

SDL processes may only create instances of processes defined in the same block. When a process creates another instance of itself, a token is put in the place **STARTUP!CREATE** place. When a process creates an instance of another process **other_process**, a token is put on a place **OUTPUT!create!other_process**. This token is removed by a transition and put on the place **STARTUP!CREATE** of the process **other_process**.

4.3.3 Process Output

Processes use the output construct to send signals to other processes. If more than one signal is output in one transition, then the order in which these are specified should be preserved. Signals may be output **VIA** a specific signal route and can be addressed **TO** a specific process by giving the process identifier. The process identifiers are a data type *PId* and may be set to *SELF*, *PARENT*, *OFFSPRING* and *SENDER*. For instance, a process could send a signal to the sender of the last received signal (*SENDER*), or to the process that created it (*PARENT*).

In SDL systems several instances of a process can exist, so deciding which instance of a process to send a signals to can become very complex. A router process can then be written whose purpose is to remember the *PId*'s of all the *process instances*, and to route the signals to the appropriate *instance*. The array data structure shown below could be used to store such PId information.

```
NEWTYPE PId_Array
   ARRAY(PortRange, PId);
ENDNEWTYPE PId_Array;
```

SDL-net Mapping

A SDL-net place is created for all the signals that are output from a process. For each signal in the output signal set, a corresponding colour is created on the output place. The output place does not need to be a *queued place* because the transitions that remove the tokens from the output place are immediate, and so take priority over the *timed transitions*, thus preserving the order of the signals.

Usually in the mapping, if more than one action is to be done in a *process transition*, the actions are merged into one transition. This can affect the order in which signals are output. For example, in the process in Figure 4.26, **s1** should be sent before **s2**. In the generated SDL-net in Figure 4.27 the order is lost, and **s1** and **s2** are sent at the same time. To preserve the order, an intermediate

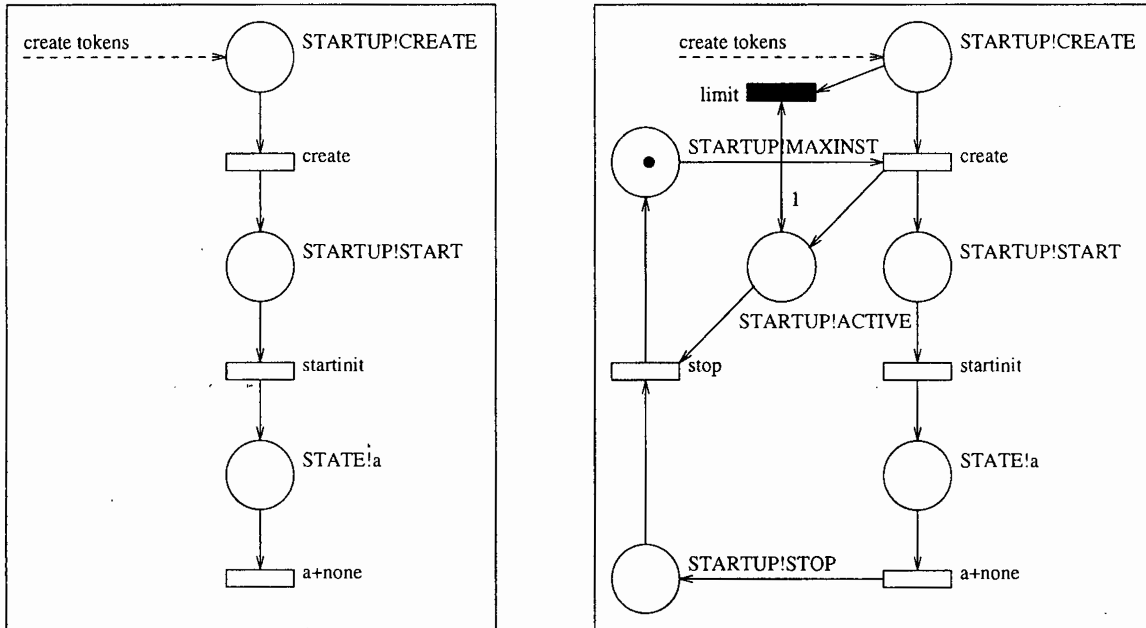

Figure 4.24: SDL-net of **PROCESS p1(0)** and **PROCESS p2(0,1)**.

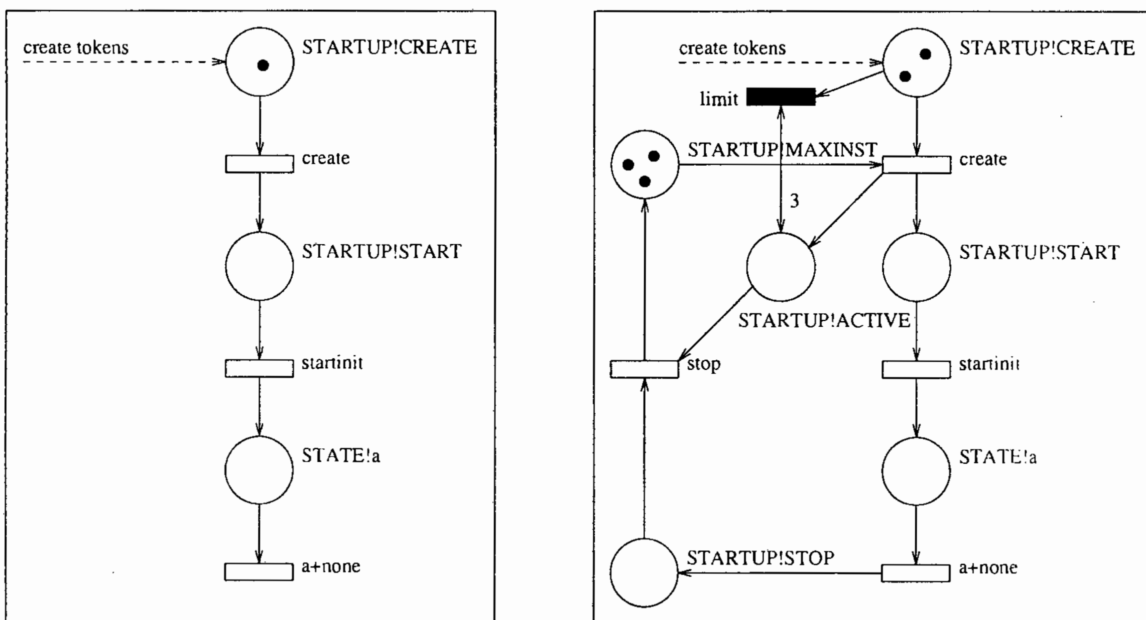

Figure 4.25: SDL-net of **PROCESS p3(1)** and **PROCESS p4(2,3)**.

transition is put inbetween the two outputs as shown in Figure 4.28. This will increase the number of tangible states in the SDL-net and should be avoided if possible. The mapping may be set with the **safe** option to translate the outputs in the original order or to disregard the order in which they are sent.

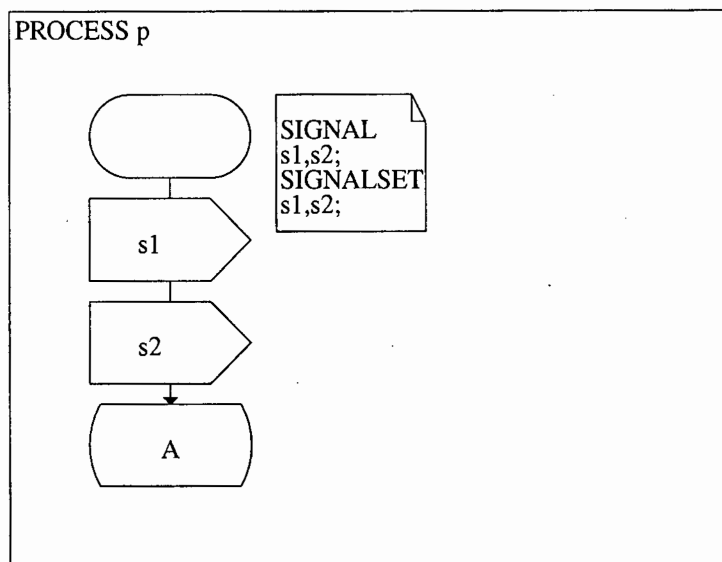

Figure 4.26: Two outputs in one transition.

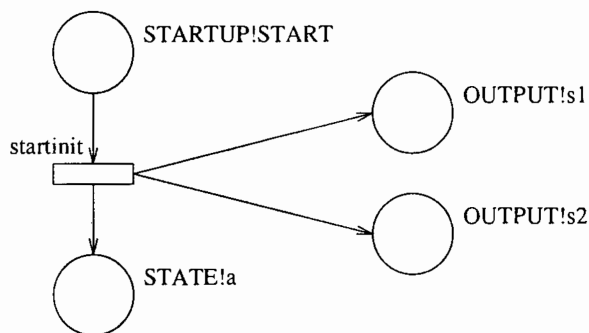

Figure 4.27: SDL-net where order of the two outputs is lost.

The process identifiers are unique numbers that are generated dynamically and that determine the process instances. In the SDL-net, managing which *process identifier* corresponds to which *process instance* would require that complex data structures, such as the *array*, are supported. Complex data structures are not supported in the SDL-net, so a signal cannot be sent to a particular *instance* of a process. Since in SDL-92 a *non-deterministic* decision is taken when multiple processes could receive a signal, the same approach is taken in this thesis.

4.3.4 Process Data

As part of the auxiliary *data types* for the EFSMs, SDL provides certain standard data structures such as *integer* and *real* numbers, *characters*, *text strings* and *arrays*. New data structures, also called *sorts*, are specified using *newtype* and operations on the data contained in the *newtype* are defined using *axioms*.

SDL allows the definition of any needed sort, subject only to the requirement that such a definition is formally specified. By contrast, for programming languages, there are implementation considerations which require that the set of available sorts be limited.

Variables are objects declared within processes which are associated with a value by assignment. The value is returned when the variable is accessed.

SDL allows a variable type to have a range specified with the *syntype* construct, which is especially useful when minimum and maximum values of a variable need to be specified.

The *synonym* construct is used to assign a constant value to a variable.

When variables are sent as parameters in signals, the type needs to be defined in the same scope as the signal. The value sent in the signal has to be a valid value for that type.

SDL-net Mapping

It is not possible to map general data types from SDL, as the SDL-net does not have the expressive power necessary for representing complex data. Data in SDL can only be represented in the SDL-net by means of a coloured place with a colour for each value of the data type. A range is then needed for the variable, as defined with *syntype*, so that it may be enumerated with all possible values. An example of a process using *Integers* is given in Figure 4.29. A SDL-net representing that SDL process is shown in Figure 4.30. Note that this is only an illustration of what the *reduced* net would look like. A full translation of data types from SDL to Petri nets is a topic to itself and not covered in this thesis.

It is possible to represent some simple SDL expressions in SDL-net. For example, if the variable **count** has a range of 1 to 4, then **count := count + 1** could be represented by the net illustrated in Figure 4.31. The undefined value is set if the data variable is not initialised, if it underflows or if it overflows. This is used to find dynamic errors in the specification. The expression **count := 3** could be represented as in Figure 4.32. The test for **count** $<$ **3** could be converted to the net shown in Figure 4.33.

This part of the translation was tested manually in [FKPP95]. A more descriptive Petri net than the SDL-net would be needed to map SDL data types automatically. Examples of such a net would

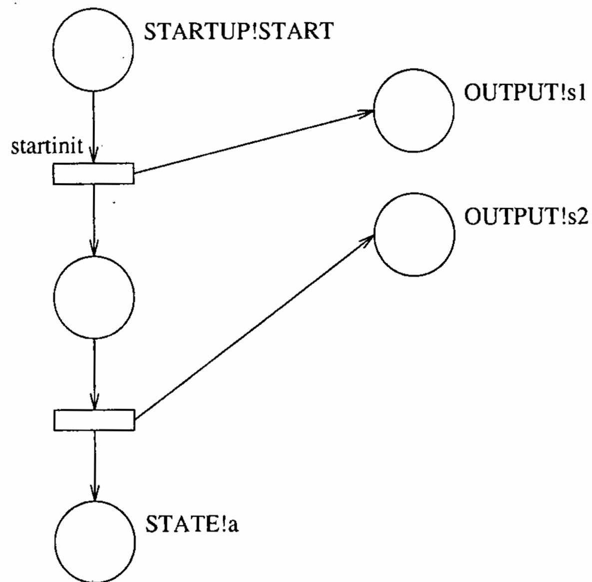

Figure 4.28: SDL-net where order of the two outputs is preserved.

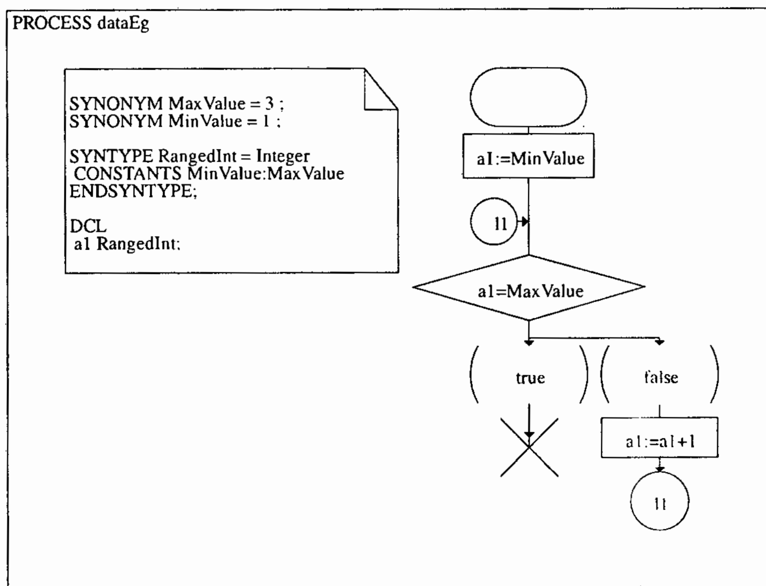

Figure 4.29: SDL Process containing variables.

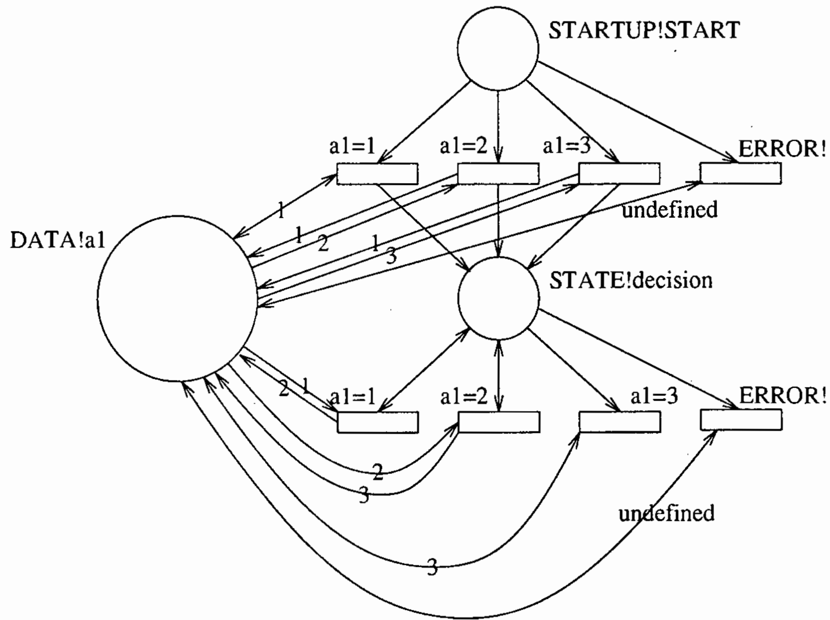

Figure 4.30: SDL-net of Figure 4.29.

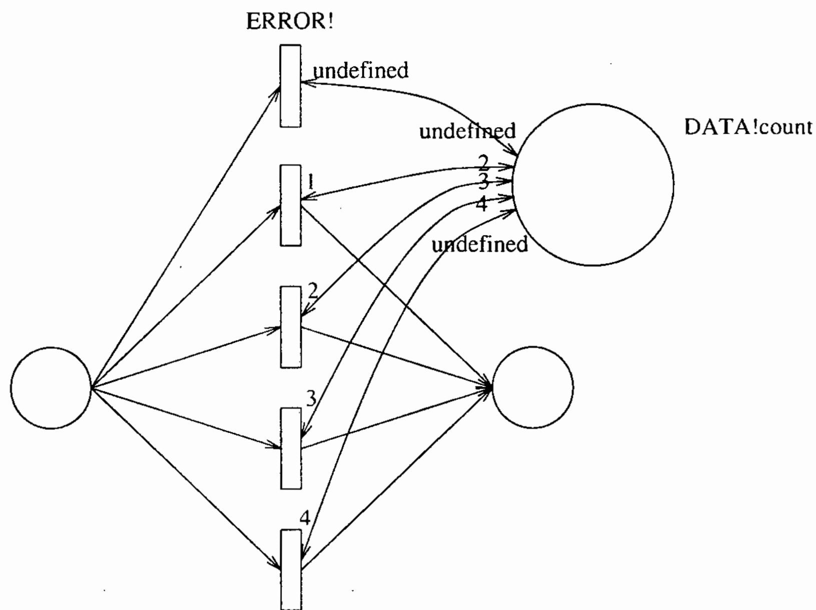

Figure 4.31: SDL-net equivalent of **count := count + 1**.

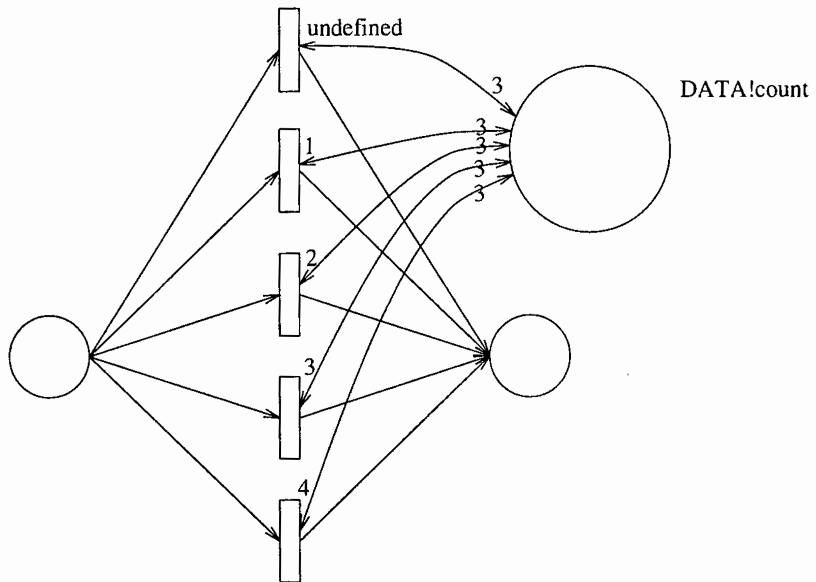

Figure 4.32: SDL-net equivalent of **count := 3**.

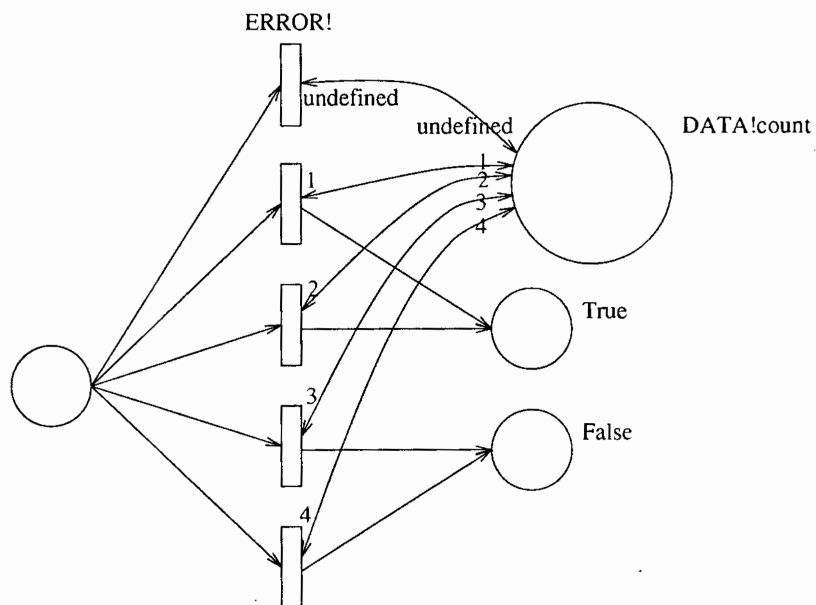

Figure 4.33: SDL-net equivalent of **count** < **3**.

be the new coloured Petri net [Jen94] that allows complex colours for the tokens, or an algebraic Petri net [DH90, Rei91] that allows axioms to be defined on the colours.

4.3.5 Process Decision

In SDL-88 process decisions had to be deterministic. SDL-92 introduced the *any* construct which is used as a non-deterministic decision. Because SDL data is not automatically mapped, only the non-deterministic *any* decisions are supported. When a deterministic decision is encountered in a process, the mapping has to generate a warning that the deterministic decision is not supported and that it will be replaced by a non-deterministic decision.

The transition branches out at a decision. If the decision ends before all the branches are terminated, then the remaining transition branches are joined again.

Figure 4.34 illustrates a SDL process with one non-deterministic decision (**any**) and one deterministic decision (**count**).

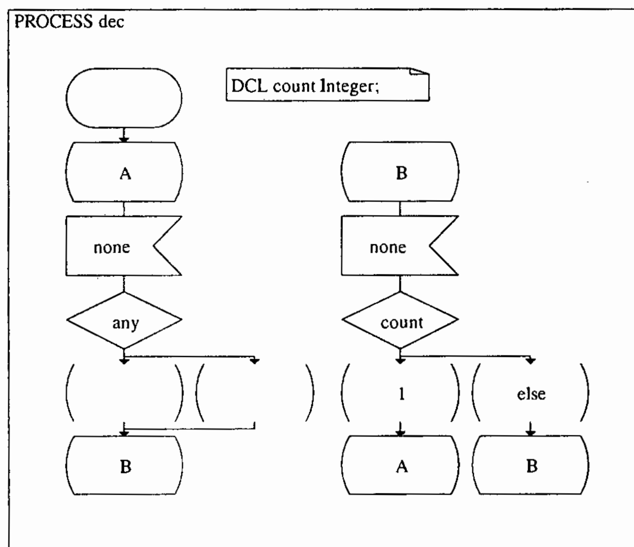

Figure 4.34: SDL Process with non-determinstic (**any**) decision and deterministic (**count**) decision.

SDL-net Mapping

A decision is mapped to a place in the SDL-net. If the transition control joins again as it does in the non-deterministic decision in Figure 4.34, an end-decision place is created in the SDL-net so the transition branches can join again.

The SDL-net mapping of Figure 4.34 is shown in Figure 4.35. Each decision branch is mapped to a transition. The delay of the decision is added to each of the answer transitions and not to

the transition before the decision. It is possible to get the situation that one of the branches of the decision has a delay of 0, and is thus an immediate transition, and that the other branch has some timing delay. If this happens, then the immediate transition will always take priority over the timed transition, according to the GSPN definition. It is therefore advisable to set the delay of a decision as non-zero.

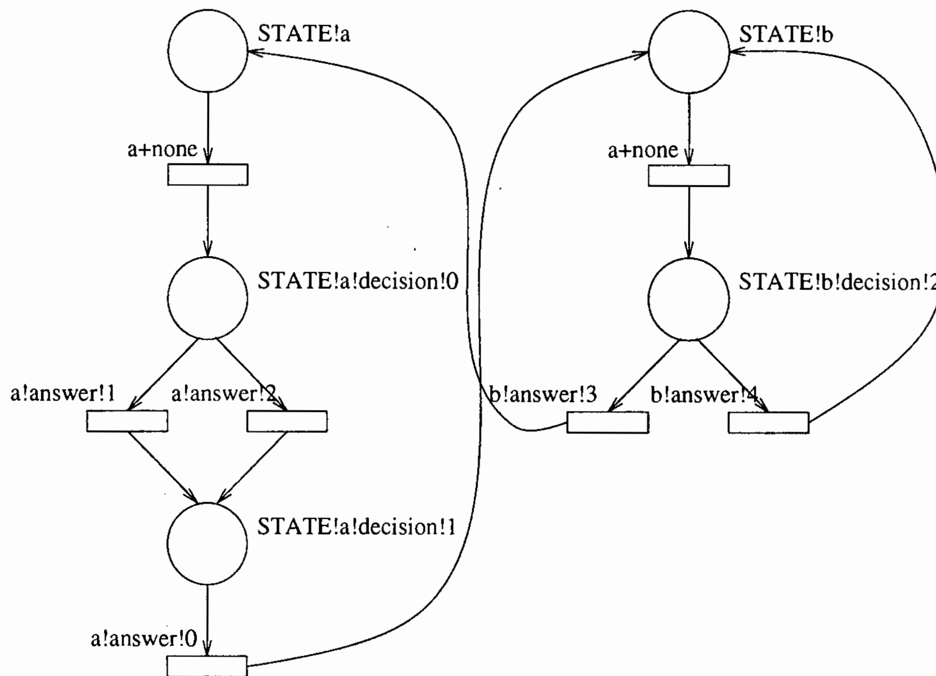

Figure 4.35: SDL-net equivalent of Figure 4.33.

4.4 Summary of Chapter

To summarise this chapter a diagram is included in Figure 4.36 that shows which parts of SDL are included in the mapping and which are not directly supported. Note that [IT93b] shows transformations with which many of these unsupported constructs can be mapped to the supported constructs.

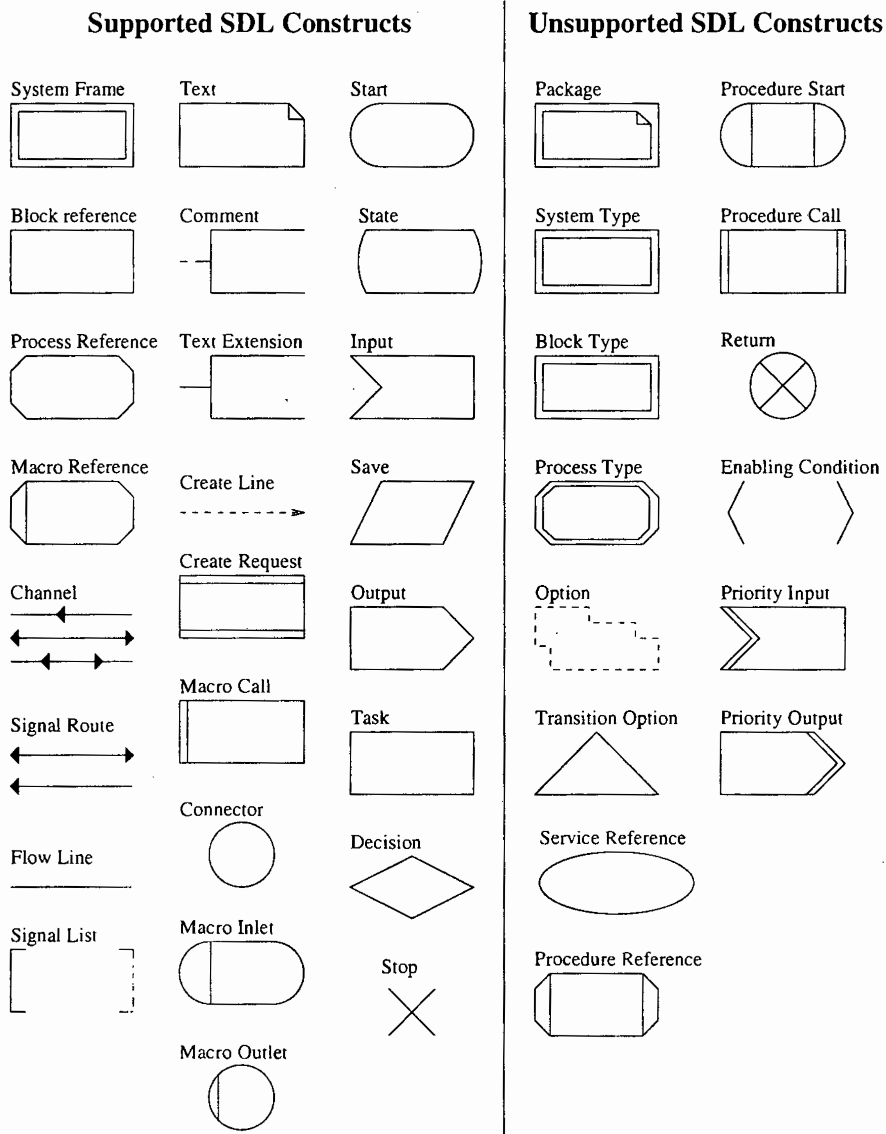

Figure 4.36: This diagram illustrates which SDL constructs are supported by this analysis method.

Chapter 5

Definition of SDL-net

In this chapter basic Petri net definitions are given which are used to define the SDL-net. The queueing Petri net with queued place is formally defined and three new queued places are introduced that are used specifically to analyse SDL systems. The SDL-net is formally specified and the analysis of the SDL-net is described.

5.1 Basic Definitions

The Petri nets defined in this section include untimed nets, such as the coloured Petri net (CPN), and timed nets, such as the generalized stochastic Petri net (GSPN), the coloured generalized stochastic Petri net (CGSPN) and the queueing Petri net (QPN). These nets are defined below as a basis for the definition of the SDL-net.

5.1.1 Coloured Petri net

The following four definitions are from [BK96]:

Definition 1 *A* multi-set *m, over a non-empty set S, is a function* $m \in [S \mapsto \mathbb{N}_0]$. *The non-negative integer m(s)* $\in \mathbb{N}_0$ *is the number of appearances of the element s in the multi-set m.*

Definition 2 Addition of multi-sets *is defined as* $\forall m_1, m_2 \in S_{MS}$ *define* $(m_1 + m_2)(s) := m_1(s) + m_2(s)$.

Definition 3 *A coloured Petri net (CPN) is a 6-tuple* $CPN = (P, T, C, I^-, I^+, M_0)$, *where*

- P *is a finite and non-empty set of* places,

- T *is a finite and non-empty set of* transitions,
- $P \cap T = \emptyset$,
- C *is a* colour function *defined from* $P \cup T$ *into finite and non-empty sets,*
- I^- *and* I^+ *are the* backward *and* forward incidence functions *defined on* $P \times T \ni I^-(p,t)$, $I^+(p,t) \in [C(t) \mapsto C(p)_{MS}], \forall (p,t) \in P \times T$.
- M_0 *is a function defined on P describing the* initial marking $\ni M_0(p) \in C(P)_{MS}, \forall p \in P$.

5.1.2 Generalized stochastic Petri net

Definition 4 *A* generalized stochastic Petri net *(GSPN) is a 4-tuple* $GSPN = (PN, T_1, T_2, W)$, *where*

- $PN = (P, T, I^-, I^+, M_0)$ *is the underlying* Place-Transition net *[BK96]*
- $T_1 \subseteq T$ *is the set of* timed *transitions,* $T_1 \neq \emptyset$,
- $T_2 \subset T$ *is the set of* immediate *transitions,* $T_1 \cap T_2 = \emptyset$, $T = T_1 \cup T_2$
- $W = (w_1, \ldots, w_{|T|})$ *is an array whose entry* w_i
  - *is a rate* $\in \mathbb{R}^+$ *of a negative exponential distribution specifying the firing delay, when transition* t_i *is a timed transition, i.e.* $t_i \in T_1$ *or*
  - *is a weight* $\in \mathbb{R}^+$ *specifying the relative firing frequency, when transition* t_i *is an immediate transition, i.e.* $t_i \in T_2$.

5.1.3 Coloured generalised stochastic Petri net

Definition 5 CGSPN *A* coloured generalised stochastic Petri net *(CGSPN) is a 7-tuple* $CGSPN = (P, T, C, I^-, I^+, M_0, W)$, *where*

- P *is a finite and non-empty set of places,*
- $T = T_1 \cup T_2$ *where*
  - $T_1 \subseteq T$ *is the set of timed transitions,* $T_1 \neq \emptyset$,
  - $T_2 \subset T$ *denotes the set of immediate transitions,* $T_1 \cap T_2 = \emptyset$,
- $P \cap T = \emptyset$.

- *C is a colour function defined from $P \cup T$ into finite and non-empty sets,*
- *I^- and I^+ are the backward and forward incidence functions defined on $P \times T \ni I^-(p,t)$, $I^+(p,t) \in [C(t) \rightarrow C(p)_{MS}], \forall\, (p,t) \in P \times T$ and $C(p)_{MS}$ the set of all finite multi-sets over $C(p)$,*
- *M_0 is a function defined on P describing the initial marking $\ni M_0(p) \in C(p)_{MS}, \forall p \in P$.*
- *$W = (w_1, \ldots, w_{|T|})$, $w_i \in \mathbb{R}^+$ is a weight function which assigns an exponential rate to the colour set of a timed transition and a colour dependent firing frequency to the colour set of the immediate transitions.*

5.1.4 Queueing Petri net based on CGSPN

Definition 6 *From [BK96]:*
A queueing Petri net *(QPN) is a 3-tuple $QPN = (CGSPN, P_1, P_2)$, where*

- *CGSPN is the underlying* coloured GSPN,
- *$P_1 \subseteq P$ is the set of* queued places *and*
- *$P_2 \subseteq P$ is the set of* ordinary places, *$P_1 \cap P_2 = \emptyset$, $P = P_1 \cup P_2$.*

A queued place, as illustrated in Figure 5.1, consists of two components: the queue and a depository for tokens which have completed their service at the queue. Tokens, when fired onto a queued place by any of the input transitions of the place, are inserted into the queue according to the scheduling strategy of the queue. Tokens in a queue are not available for the transitions. After completion of their service, the tokens are placed onto the depository from where they are available to all output transitions of the queued place. An enabled timed transition will fire after an exponentially distributed time delay and an immediate transition fires immediately as for CGSPNs.

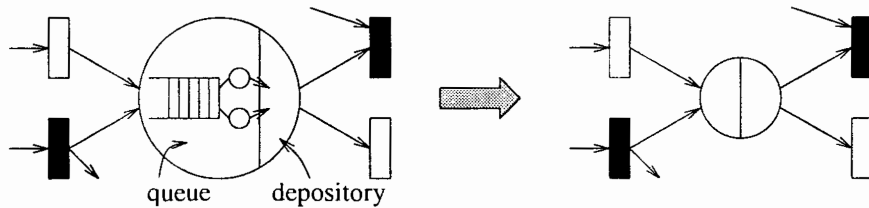

Figure 5.1: A queued place and its shorthand notation

Definition 7 Queue *A queue is a 3-tuple $QU(p) = (ST(p), TR(p), AC(p))$, where*

$ST(p)$ *specifies the feasible states of a queue.*

$TR(p) : ST(p) \rightarrow ST(p)$ *specifies what occurs when a token is served. Note that this function may be used to change the state in many different ways, and does not only have to result in a token being put into the depository.*

$AC(p) : ST(p) \times S(p) \rightarrow ST(p)$ *specifies the next state of the queue after a token from the colourset* $S(p)$ *arrives at the place.*

Subscripts will be used In the definitions of the transitions to distinguish between the various transitions. For example, $TR(p)_{SP,1}$ will represent the first service transition of the Save-place.

5.2 New Queued Places

The three new queued places, namely the Save-place, the Timer-place and the Channel-place, discussed in previous sections, are formally defined in this section.

5.2.1 Save-place

A Save-place of a SDL-net is a special place which models the FIFO *service discipline* in the signal queues, and includes the semantics of the SAVE construct in SDL. Tokens in signal queues experience no delays, but they rather experience a type of blocking which arises from the FIFO *service discipline.*

The weight of the immediate transitions defined in the Save-place is given by α.

Formal description of the Save-place

Let Q_{SP} represent the set of all Save-places in the net, $C(p)$ the set of all coloured tokens representing signals that can arrive at Save-place p and $\bar{C}(p)$ the set of all coloured tokens representing *control* signals that can arrive at Save-place p. Let $S(p) = C(p) \cup \bar{C}(p)$ be the set of all coloured tokens that can arrive at the Save-place p. In addition, let $C(p) \cap \bar{C}(p) = \emptyset$.

Define $\sigma : S(p) \rightarrow S(p)$ to be a bijective function that maps a signal token to its control signal token, and vice versa. So, if for some Save-place p, $a \in C(p)$, $@a \in \bar{C}(p)$ and $@a$ is the control token that saves signal a, then $\sigma(a) = @a$ and $\sigma(@a) = a$.

Definition 8 Feasible state of the queue *For a Save-place p, let* $s_n, \ldots, s_1$ *be a sequence of coloured tokens with colours from the set* $C(p)$, s_1 *being at the head of the queue. The set of possible arrangements* $s_n, \ldots, s_1$ $\forall n \in \mathbb{N}$, *constitute the set* $Q(p)$ *of feasible states* $s_n, \ldots, s_1$ *of the queue for Save-place p.*

Definition 9 Feasible marking of the available signal *Let ϵ represents the empty token. Let $a \in \mathcal{C}(p) \cup \epsilon$ represent the currently available signal.*

Definition 10 Feasible marking of the depository *Let $d : \mathcal{S}(p) \to \mathbb{N}_0$ be a multi-set over $\mathcal{S}(p)$ and let $\mathcal{D}(p)$ be the set of all such multi-sets, d is called a feasible marking of the depository.*

Definition 11 State descriptor of a Save-place *Let $\mathcal{Q}(p)$ be the set of feasible states of the queue, a the currently available signal and $\mathcal{D}(p)$ the set of feasible markings of the depository. We write $((s_n, \ldots, s_1), a, d) \subseteq \mathcal{Q}(p) \times (\mathcal{C}(p) \cup \epsilon) \times \mathcal{D}(p)$ for the state descriptor of the Save-place p. This state descriptor is denoted by $ST(p)$.*

Definition 12 State transitions of a Save-place, p

- *On arrival of a token s at Save-place p*

  Let $AC(p) : ST(p) \times \mathcal{S}(p) \to ST(p)$ be a function that specifies the change in state of the Save-place p when a token s arrives. $AC(p)$ is defined by the following transitions:

  $AC_{SP,1}(p) : (((s_n, \ldots, s_1), a, d), s) \to ((s, s_n, \ldots, s_1), a, d)$
  with weight α
  when $s \in \mathcal{C}(p)$

  $AC_{SP,2}(p) : (((s_n, \ldots, s_1), a, d), s) \to ((s_n, \ldots, s_1), a, (d + \delta(s)))$
  with weight α
  when $s \in \bar{\mathcal{C}}(p)$

  where $\delta(s)$ is the Kronecker function [Bau93].

- *When a token is served at the Save-place p*

  Let $TR(p) : ST(p) \to ST(p)$ be a function that specifies the change in state of the Save-place p when a token is served. $TR(p)$ is defined by the following transitions:

  $TR_{SP,1}(p) : ((s_n, \ldots, s_1), a, d) \to ((s_n, \ldots, s_1), \epsilon, (d - \delta(\sigma(s))))$
  with weight α
  if $\exists s \in \bar{\mathcal{C}}(p) \ni d(s) > 0$ and $d(\sigma(s)) > 0$

  $TR_{SP,2}(p) : ((s_n, \ldots, s_{i+1}, s_i, s_{i-1}, \ldots, s_1), a, d) \to ((s_n, \ldots, s_{i+1}, s_{i-1}, \ldots, s_1), \epsilon, d)$
  with weight α
  if $a = s_i$ and $d(s_i) = 0$ and $\forall k < i, s_k \neq s_i$

  $TR_{SP,3}(p) : ((s_n, \ldots, s_1), a, d) \to ((s_n, \ldots, s_1), \epsilon, (d - \delta(a)))$
  with weight α
  if $d(\sigma(s_i)) = 0$ and $a \neq \epsilon$ and $a \neq s_i$ and $\forall k < n, d(\sigma(s_k)) > 0$

$$TR_{SP,4}(p) : ((s_n, \ldots, s_1), a, d) \rightarrow ((s_n, \ldots, s_1), s_i, (d + \delta(s_i)))$$
with weight α
$$\text{if } d(\sigma(s_i)) = 0 \text{ and } a = \epsilon \text{ and } \forall k < n, d(\sigma(s_k)) > 0$$

These state transitions are illustrated graphically in Figure 5.2. Each of the solid arrows represents an immediate transitions.

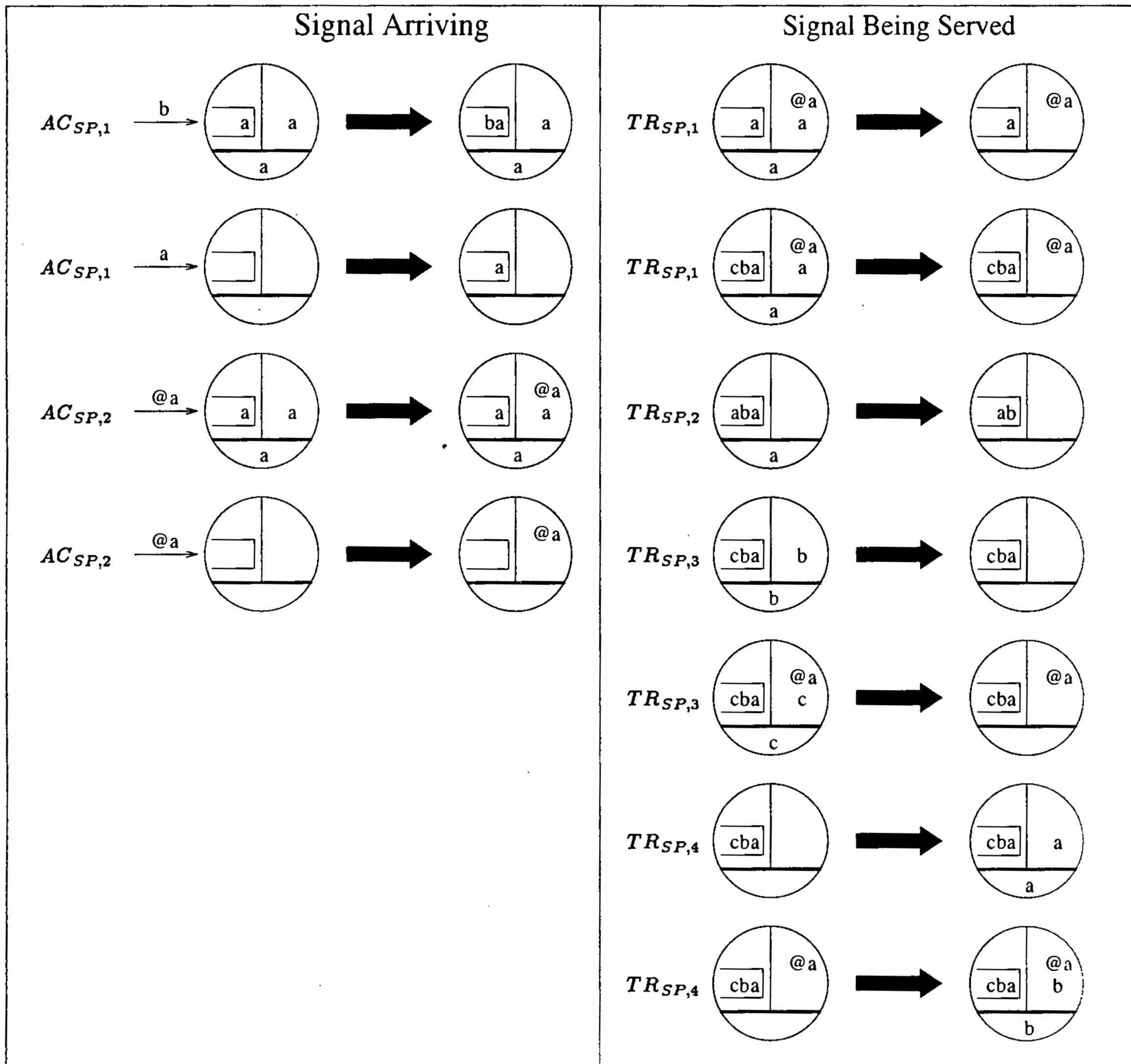

Figure 5.2: Transitions in the Save-place

5.2.2 Timer-place

A Timer-place of a SDL-net is a special place which models the deterministic timing delay of the *timer* construct in SDL using a Coxian phase-type distribution. The timer can be *reset*, in which case it needs to be stopped. A special reset token is introduced that can eject the Coxian timer. It

was chosen to only allow one timer per Timer-place. This need not be the case, but was chosen to conceptually simplify the resulting net.

The weight of the immediate transitions defined in the Save-place is given by α.

Formal description of the Timer-place

Let Q_{TP} represent the set of all Timer-places in the net. For each SDL timer in the system a separate Timer-place is created. Let $\mathcal{C}(p)$ be the set containing the token representing the setting of an SDL timer in an SDL process. Let $\bar{\mathcal{C}}(p)$ be the set containing only the token representing the reset token of the SDL timer. Let $\mathcal{S}(p) = \mathcal{C}(p) \cup \bar{\mathcal{C}}(p)$ be the set of coloured tokens arriving at the Timer-place. In addition, $\mathcal{C}(p) \cap \bar{\mathcal{C}}(p) = \emptyset$.

Let $\varrho : \mathcal{S}(p) \to \mathcal{S}(p)$ be a bijective function that maps a timer token to its reset token, and vice versa. So, if for Timer-place p, $t \in \mathcal{C}(p)$, $\#t \in \bar{\mathcal{C}}(p)$, and $\#t$ is the reset token of t, then $\varrho(t) = \#t$ and $\varrho(\#t) = t$.

Let $k(p)$ represent the maximum number of phases for Timer-place p.

Definition 13 Rates and probabilities in Coxian server. *Let $\mu(p) : \mathbb{N}_0 \to \mathbb{R}_{(0,\infty)}$ be a function that specifies the mean rate of each phase in the Coxian server. Thus $\mu(p)(n)$ represents the rate of phase n of Timer-place p.*

Let $\alpha(p) : \mathbb{N}_0 \to \mathbb{R}_{[0,1]}$ be a function that specifies the probability of moving from a Coxian phase to the next. Thus $\alpha(p)(n)$ represents the probability of going from phase n to phase $n+1$ in Timer-place p. Hence $1 - \alpha(p)(n)$ is the probability of leaving the Coxian server after phase n.

Definition 14 Feasible marking of the current Coxian phase. *Let $n \in \mathbb{N}_0$ indicate the current Coxian phase. As there is only one timer per Timer-place, no further information is needed.*

Definition 15 Feasible marking of the depository. *Let $d : \mathcal{S}(p) \to \mathbb{N}_0$ be a multi-set over $\mathcal{S}(p)$ and let $\mathcal{D}(p)$ be the set of all such multi-sets, d is called a feasible marking of the depository.*

Definition 16 State descriptor of a Timer-place *For a Timer-place p, let $\mathbb{N}_0$ be the set of feasible states of the phases and $\mathcal{D}(p)$ the set of feasible markings of the depository. We write $(t, d) \subseteq \mathbb{N}_0 \times \mathcal{D}(p)$ for the state descriptor of the Timer-place p. This state descriptor is denoted by $ST(p)$.*

Definition 17 State transitions of a Timer-place p

- *On arrival of a token s at Timer-place p*

  Let $AC(p) : ST(p) \times \mathcal{S}(p) \rightarrow ST(p)$ *be a function that specifies the change in state of the Timer-place when a token s arrives.* $AC(p)$ *is defined by the following transitions:*

  $AC_{TP,1}(p) : ((m, d), s) \rightarrow (1, d)$
  with weight α
  if $s \in \mathcal{C}(p)$ *and* $m = 0$

  $AC_{TP,2}(p) : ((m, d), s) \rightarrow (m, (d + \delta(s)))$
  with weight α
  if $s \in \bar{\mathcal{C}}(p)$ *or*
  $s \in \mathcal{C}(p)$ *and* $m > 0$

- *When a token is served at the Timer-place p*

  Let $TR(p) : ST(p) \rightarrow ST(p)$ *be a function that specifies the change in state of the Timer-place when a token is served.* $TR(p)$ *is defined by the following transitions:*

  $TR_{TP,1}(p) : (m, d) \rightarrow (0, (d + \delta(s)))$
  with weight α
  if $m > 0$ *and* $d(\varrho(s)) > 0$

  $TR_{TP,2}(p) : (m, d) \rightarrow (m + 1, d)$
  at rate $\alpha(p)(m) \cdot \mu(p)(m)$
  if $m > 0$ *and* $m < k(p)$

  $TR_{TP,3}(p) : (m, d) \rightarrow (0, (d + \delta(s)))$
  at rate $(1 - \alpha(p)(m)) \cdot \mu(p)(m)$
  if $m > 0$ *and* $m < k(p)$ *and* $s \in \mathcal{C}(p)$

  $TR_{TP,4}(p) : (m, d) \rightarrow (0, (d + \delta(s)))$
  at rate $\mu(p)(m)$
  if $m = k(p)$ *and* $s \in \mathcal{C}(p)$

These state transitions are illustrated graphically in Figure 5.3. The solid arrows represent immediate transitions and the outlined arrows timed transitions.

5.2.3 Channel-place

A Channel-place of a SDL-net is a special place which models the FIFO service discipline while approximating the general channel delay time distribution with a Coxian phase-type distribution.

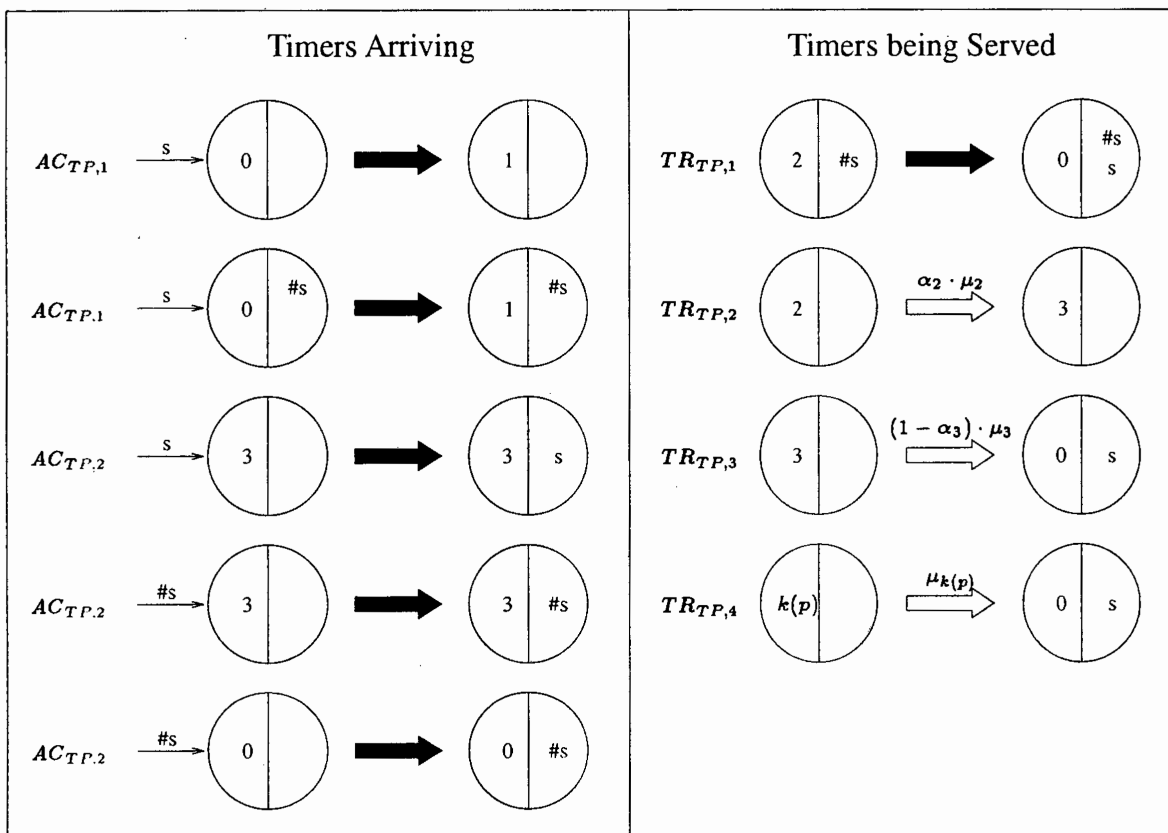

Figure 5.3: Transitions in the Timer-place

Formal description of the Channel-place

Let Q_{SP} represent the set of all Timer-places in the net. Let $\mathcal{S}(p)$ be the set of all coloured tokens representing SDL signals arriving at the Channel-place p.

Let $k(p)$ represent the maximum number of phases for Channel-place p.

Definition 18 Feasible state of the queue. *Let $s_n, \ldots, s_1$ be a sequence of coloured tokens with colours from the set $\mathcal{S}(p)$, s_1 being at the head of the queue. The set of all possible arrangements $s_n, \ldots, s_1$ for all $n \in \mathbb{N}$ constitute the set $\mathcal{Q}(p)$ of feasible states $s_n, \ldots, s_1$ of the queue.*

Definition 19 Rates and probabilities in Coxian server. *Let $\mu(p) : \mathbb{N}_0 \rightarrow \mathbb{R}_{(0,\infty)}$ be a function that specifies the rates of the phases in the Coxian server. Thus $\mu(p)(n)$ represents the rate of phase n in place p.*

Let $\alpha(p) : \mathbb{N}_0 \rightarrow \mathbb{R}_{[0,1]}$ be a function that specifies the probability of moving from one Coxian phase to the next. Thus $\alpha(p)(n)$ represents the probability of going from phase n to phase $n+1$ in place p. Hence $1 - \alpha(p)(n)$ is the probability of leaving the Coxian server.

Definition 20 Feasible marking of the Coxian phase *Let ϵ represent an empty channel. Let $a \in \mathcal{C}(p) \cup \epsilon$ represent the signal currently on the channel. Furthermore, let $\mathcal{T}(p) = (\mathcal{C}(p) \cup \epsilon) \times \mathbb{N}_0$ be the set of all possible values for the Coxian phase, where $\mathbb{N}_0$ covers the number of the current Coxian phase.*

Definition 21 Feasible marking of the depository. *Let $d : \mathcal{S}(p) \rightarrow \mathbb{N}_0$ be a multi-set over $\mathcal{S}(p)$ and let $\mathcal{D}(p)$ be the set of all such multi-sets; d is called a feasible marking of the depository.*

Definition 22 State descriptor of Channel-place p *Let $\mathcal{Q}(p)$ be the set of feasible states of the queue, $\mathcal{T}(p)$ the set of feasible states of the Coxian server and $\mathcal{D}(p)$ the set of feasible markings of the depository. We write $((s_n, \ldots, s_1), (a, n), d) \subseteq \mathcal{Q}(p) \times \mathcal{T}(p) \times \mathcal{D}(p)$ for the state descriptor of the Channel-place p. This state descriptor is denoted by $ST(p)$.*

Definition 23 State transitions of a Channel-place p

- *On arrival of a token s at Channel-place p*

  Let $AC(p) : ST(p) \times \mathcal{S}(p) \rightarrow ST(p)$ be a function that specifies the change in state of the Channel-place p when a token s arrives. $AC(p)$ is defined by the following transitions:

  $AC_{CP,1}(p) : (((s_n, \ldots, s_1), (a, m), d), s) \rightarrow ((s, s_n, \ldots, s_1), (a, m), d)$
  with weight α

- *When a token is served at the Channel-place p*

  Let $TR(p): ST(p) \to ST(p)$ *be a function that specifies the change in state of the Channel-place p when a token is served.* $TR(p)$ *is defined by the following transitions:*

  $TR_{CP,1}(p): ((s_n, \ldots, s_2, s_1), (a, m), d) \to ((s_n, \ldots, s_2), (s_1, 1), d)$
  with weight α
  if $m = 0$

  $TR_{CP,2}(p): ((s_n, \ldots, s_1), (a, m), d) \to ((s_n, \ldots, s_1), (a, m+1), d)$
  at rate $\alpha(p)(m) \cdot \mu(p)(m)$
  if $m > 0$ *and* $m < k(p)$

  $TR_{CP,3}(p): ((s_n, \ldots, s_1), (a, m), d) \to ((s_n, \ldots, s_1), (\epsilon, 0), (d + \delta(a)))$
  at rate $(1 - \alpha(p)(m)) \cdot \mu(p)(m)$
  if $m > 0$ *and* $m < k(p)$

  $TR_{CP,4}(p): ((s_n, \ldots, s_1), (a, m), d) \to ((s_n, \ldots, s_1), (\epsilon, 0), (d + \delta(a)))$
  at rate $\mu(p)(m)$
  if $m = k(p)$

These state transitions are illustrated graphically in Figure 5.4. The solid arrows represent immediate transitions and the outlined arrows timed transitions.

5.3 SDL-net

Definition 24 SDL-net *With the above definitions, one can now define a SDL-net* ($SDLN$) *as a coloured generalised stochastic Petri net, which includes places that are of the type Save-place, Timer-place and Channel-place. That is,* $SDLN = (CGSPN, Q)$ *where*

- $CGSPN = (P, T, C, I^-, I^+, M_0, W)$ *is the underlying coloured generalised stochastic Petri net,*
- $Q = (Q_{SP}, Q_{TP}, Q_{SP}, (q_1, \ldots, q_{|P|}))$ *where*
  - $Q_{SP} \subseteq P$ *is the set of Save-places,*
  - $Q_{TP} \subseteq P$ *is the set of Timer-places,*
  - $Q_{CP} \subseteq P$ *is the set of Channel-places,*
  - $Q_{SP} \cap Q_{TP} = \emptyset$, $Q_{SP} \cap Q_{CP} = \emptyset$, $Q_{TP} \cap Q_{CP} = \emptyset$ *and*

$$q_i = \begin{cases} QU(p_i) & \text{if } p_i \in Q_{SP} \cup Q_{TP} \cup Q_{CP} \\ 0 & \text{otherwise} \end{cases}$$

5.4 Conclusion

In this section the SDL-net was formally defined, including the state descriptor and the state transitions. This formal definition leads directly to correctness analysis as described in Section 3.2.1 and in [Bau93]. In addition, a Markov chain can be derived from an SDL-net by considering the formal definitions. Various tools have been produced to analyse Concurrent Communicating System in this way and these will be described in the following chapter.

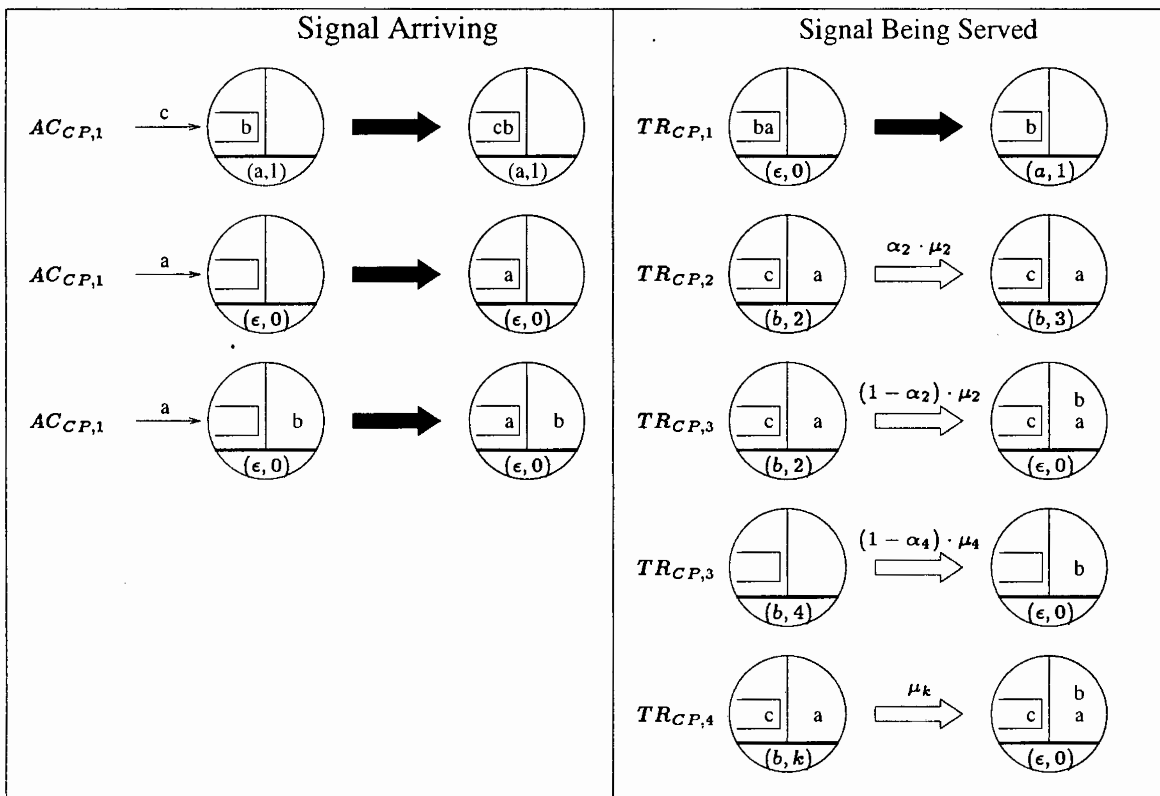

Figure 5.4: Transitions in the Channel-place

This page intentionally left blank.

Chapter 6

DNAsty

This chapter will build on the the theory and methodology established earlier in the thesis and will show that this process could be automated and applied in practice. A toolset called **DNAsty** was constructed to investigate if the SDL-net could be used for analysing Concurrent Communicating System. As explained in Section 3.1 on page 9, SDL-net formed part of a much larger CASE toolset for the specification and analysis of Concurrent Communicating System. Wherever possible, existing components were used to construct the toolset, thus illustrating that the SDL-net can be analysed with existing techniques.

The SDL toolset is illustrated in Figure 6.1. The tool modules are shown as ellipses and the interactions between the various modules are shown as rectangular, shaded boxes. The dashed-line ellipses represent modules written by third parties. Only a brief overview of the modules is given here as they will be explained in greater detail in the latter part of this chapter.

At the highest level of the toolset is the **SDLite** graphical SDL editor [Kab96], which is a graphical editor for the specification of Concurrent Communicating Systems in SDL, including methods for specifying the performance requirements, later used in the analysis. The graphical SDL representation (SDL/GR) is transformed into the equivalent textual representation (SDL/PR) using the **Translator** [Kab96]. The SDL Application Programming Interface (SDL API) [Ver95], from the company Verilog in France, is used to parse the SDL/PR, to verify that it is syntactically and semantically correct and to produce a data structure similar to the SDL Abstract Syntax Tree (SDL/AST). The tool **sdl2pn** accesses the data structure to produce an equivalent SDL-net, taking into account performance requirements specified by the user through SDLite. The SDL-net is then processed through **wam2mod**, which verifies its correctness using either Petri net invariant analysis [KBA94, ABKK95] or state space exploration, and uses the performance requirements from SDLite in deriving a Markov chain. This Markov chain is solved using DNAmaca [Kno96] which produces the required results. These results are then parsed, using the tool **mod2sdl**, and results

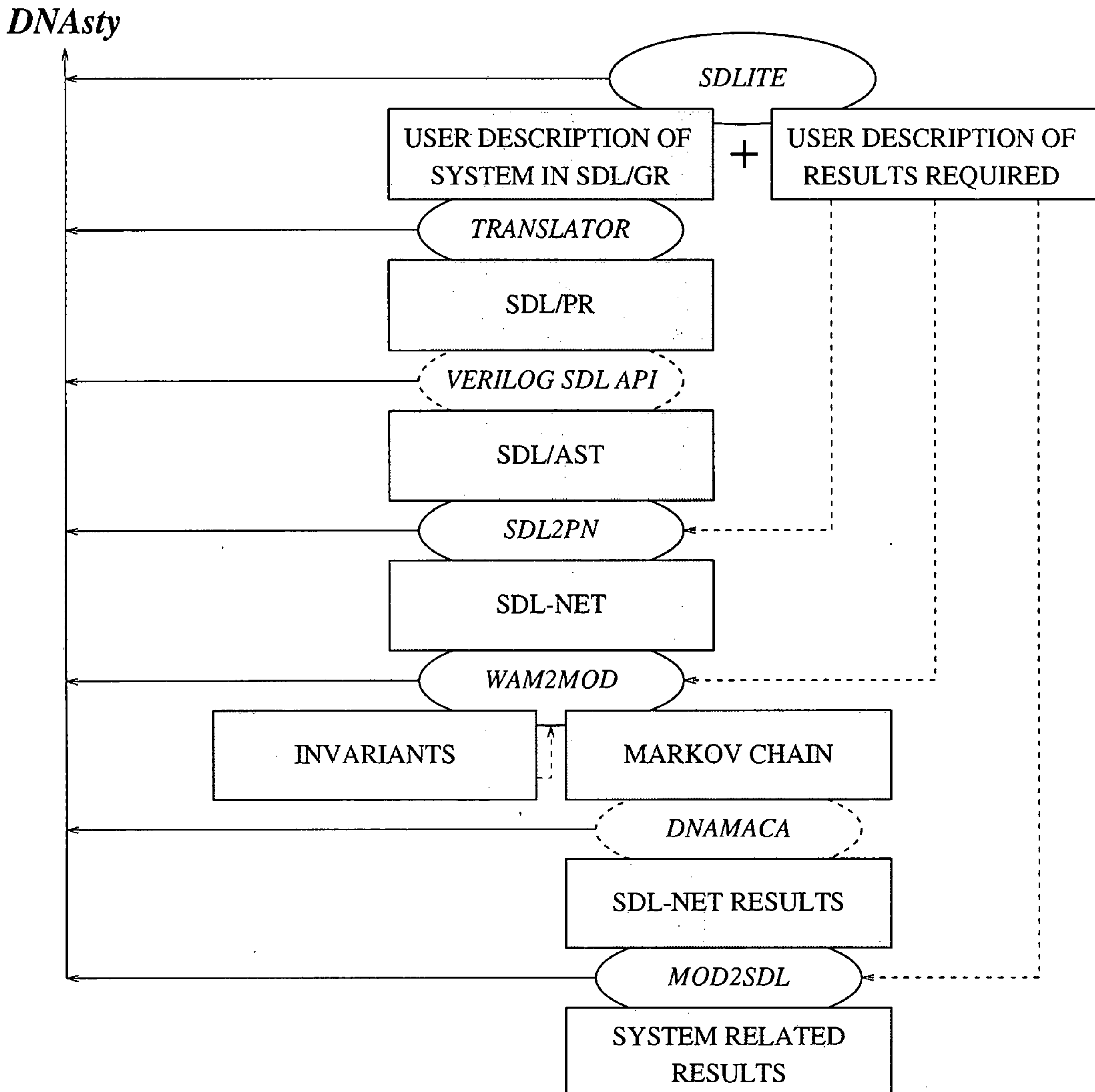

Figure 6.1: Toolset implemented to verify SDL-net analysis technique.

are produced which are applicable to the Concurrent Communicating Systems specified in SDL. To avoid unnecessary information being displayed, the user can specify which results should be shown.

6.1 SDLite

A graphical SDL/GR editor called **SDLite** [Kab96] was written as part of this thesis to provide a user-friendly interface for specifying and designing a Concurrent Communicating System. This editor was written for the Solaris 2.x X/Motif operating system using the portable C++ Application Framework **zApp** [Rog95] and was ported to the Microsoft Windows 3.1 and Microsoft Windows 95/NT operating systems.

A SDL/GR system may be converted to the equivalent SDL/PR system, and, likewise, a SDL/PR system may be parsed and converted to the equivalent SDL/GR system. When the SDL-net functional correctness and performance analysers are launched from within SDLite, as illustrated in Figure 6.2, the SDL/GR system is automatically translated to a SDL/PR system for input to the relevant tools. The SPECS simulator [BMSK96] may also be started from within SDLite using the translated SDL/PR as input.

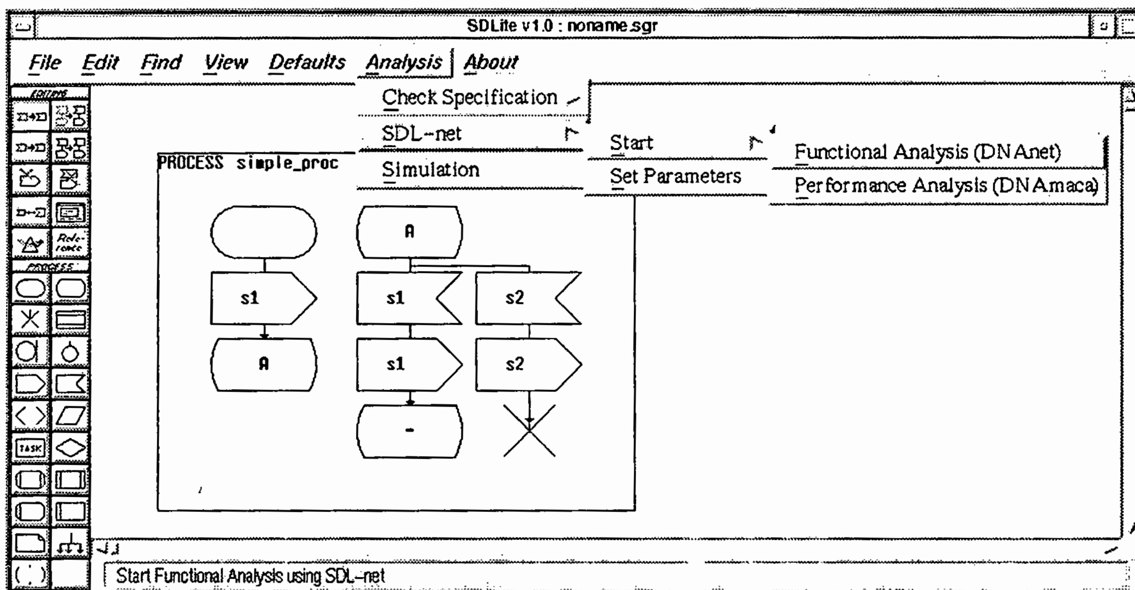

Figure 6.2: SDLite used to specify a SDL/GR system

Timing information for a channel, as described in Section 4.2.3 on page 35, may be specified directly in SDLite, this includes the number of phases in the Coxian distribution, the mean rate of each phase and the probability of its going to the next phase. An untimed channel may also be selected with a "nodelay" switch.

The SDL/GR may be exported in various graphical formats and the version of SDLite for Solaris/Motif allows the SDL/GR to be exported to the X/Windows XFig editor. The SDLite version for MS Windows, on the other hand, allows the SDL/GR to be exported as a bitmap to the clipboard, from where it may be included in other documents. All the SDL diagrams in this thesis were produced by exporting the SDL/GR to the XFig format.

SDLite does dynamic syntax checking while the system is being designed. Additional semantic and syntax validation may be done by calling the Geode checker developed by Verilog [Ver96b] from within SDLite.

6.2 SDL API and Sdl2pn

The Application Programming Interface for SDL (SDL API) [Ver95] is used to parse the SDL/PR and to verify that it is syntactically and semantically correct. The API produces a data structure that contains all the elements of the specified Concurrent Communicating System.

The tool **sdl2pn** uses this API to map a Concurrent Communicating System to a SDL-net. Since **sdl2pn** may be executed from the command line, other SDL editors may also be used to produce the SDL/PR.

The output that is generated gives the user an indication of the success of the mapping. When SDL constructs are encountered that cannot be mapped to the SDL-net, such as an *informal task*, then an appropriate warning message is shown, the construct is ignored, and the mapping continues at the next construct.

The following incorrect SDL process was mapped to an SDL-net:

```
PROCESS inform;
  START;
    NEXTSTATE FIRST;
  STATE FIRST;
    INPUT s1;
      TASK 'shake hands with operator';
      STOP;
ENDPROCESS;
```

This led to the following output being generated by **sdl2pn**:

```
$ sdl2pn inform.pr
SDL to Petri net Converter v0.1
Copyright 1996 Heinz M. Kabutz and University of Cape Town
----------------------------------------------------------
```

```
Parsing SDL using Verilog SDL API v1.0
SDL Application Programming Interfaces V1.0 sdl2pn (c) VERILOG 1995
"inform.pr", warning [2.4.1] line 1: Valid input signalset must be
specified in inform
"inform.pr", error [6.2] line 5: SIGNAL (or TIMER) s1 undefined
"inform.pr", (0 information)
"inform.pr", 1 warning
"inform.pr", 1 error
*** Generation aborted ***
```

The error was corrected as is shown in the following SDL process:

```
PROCESS inform;
  SIGNAL s1;
  SIGNALSET s1;
  START;
    NEXTSTATE FIRST;
  STATE FIRST;
    INPUT s1;
      TASK 'shake hands with operator';
      STOP;
ENDPROCESS;
```

This correct SDL system produced produced the following output:

```
$ sdl2pn inform.pr
SDL to Petri net Converter v0.1
...
SDL Parsed successfully
...
Reading in Delays for each SDL construct from file standard.per
...
Setting up Process inform
  Converting Startup
    Initial instances: 1
    Maximum instances: infinite
        Nextstate first
  Looking for necessary colours for SAVEPLACE and TIMERPLACE:
    SAVEPLACE!s1 (defined here)
  Looking for necessary colours for OUTPUT place:
  Converting States:
    first
  Making input transitions for each state:
    Inputs for Semantic state first:
```

```
      s1
        WARNING - Informal Task 'shake hands with operator' (Skipping)
"inform.pr", sdl2pn warning line 8: Informal Task not supported
        Nextstate first
==========================================================
1 Warnings found in Process inform
Finished Translating SDL to Petri net
----------------------------------------------------------
```

A warning message was caused by an *informal task*, but the mapping could still be completed despite this.

Each action in SDL may have a delay associated with it that represents how long such a SDL action would take. These delays are specified in a control file which may be generated automatically from within SDLite.

Table 6.1 represents the various SDL actions that may have time delays assigned to them.

Action
Input
Start
Informal Task
Assign Task
Output
Export
Create
Decision
Set
Reset
Procedure Call
Terminator (Nextstate, Stop, Join)
Label
Write

Table 6.1: Example of delays for SDL actions.

For debugging purposes, the DNAnet editor for GSPN's [KBA94, ABKK95] may be used to view the flattened structure of the SDL-net.

6.3 Wam2mod

The tool **wam2mod** is used to derive the Markov chain from a SDL-net. To derive the Markov chain the following information is required:

- The maximum number of tokens per Save-place and Channel-place to construct the signal queues.
- The Coxian distribution details for each timed channel and process timer.

A SDL-net that is covered by place invariants is also bounded [BK96], therefore the maximum tokens per place is calculated by considering the initial marking of the net. These place invariants are calculated by **wam2mod** using routines from the DNAnet GSPN toolset [KBA94, ABKK95]. However, a net may still be bounded even when it is not covered by place invariants. If this is the case, then the state space of the underlying CGSPN is explored by using the *state space generation techniques* of DNAmaca [Kno96]. As will be demonstrated in Section 7.2.2 on page 90 a SDL-net may be bounded even if the underlying CGSPN is unbounded. Because of this, it may not be possible to find the maximum number of tokens per place by exploring the state space of the underlying CGSPN, so the user may have to specify them.

Each Coxian server may have a different *mean delay time* and *coefficient of variation*, which may be used as parameters to the formulae described in Section 3.3.3 to calculate the values for the phases of the Coxian server.

Additional information is specified to control which analytical method is used by DNAmaca to solve the Markov chain. The Markov chain solver uses a *probabilistic bit hashing technique* which may omit states with a very small probability. An option in **wam2mod** is to monitor whether queues overflow during the generating of the state space. This is advisable if the maximum queue length was specified manually. Since this introduces an overhead in the calculation of the state space, it should only be used for the debugging of Concurrent Communicating System specifications.

6.4 DNAmaca

The Markov chain analyser DNAmaca [Kno96] is used to solve the Markov chain derived with **wam2mod** from the SDL-net. The Markov chain typically contains thousands of states and transitions. To avoid enumerating these explicitly, DNAmaca uses a high-level model description language to input the *state descriptor*, the effects of *transitions* between states and the *initial state*. The transitions are described by *enabling conditions*, an *action* and performance parameters such as the *rate* for a timed transition and the *weight* of an immediate transition. This high-level model description is used to generate *C++* code which, upon execution, yields the state space of the Markov chain. The model language includes parameters to control the state space generation by specifying the *maximum number of states* or the *maximum CPU time* available. Once the state

space of the Markov chain has been generated, it is solved using one of several methods for solving linear equations as determined by DNAmaca.

The model description language is also used to specify which performance measures to collect and report to the user. Typical performance results generated by DNAmaca for the **InRes** protocol, described in the next chapter, are shown below and a brief description of their relation to a Concurrent Communicating System described in SDL is included. Since these examples are merely to illustrate the relation between SDL and the results from DNAmaca, no knowledge of the **InRes** protocol is needed at this stage.

The following shows the distribution of colourless tokens in the SDL process state defined by "**system inres / block responder / process responder / state disconnected**". This represents the time that the process **responder** spends in the state **disconnected.**

```
State Measure 'Mean tokens on place
 main.sinres.bresponder.presponder.STATE!disconnected'
  mean                 2.7246787734e-01
  variance             1.9822913316e-01
  distribution
                  0    7.2753212266e-01
                  1    2.7246787734e-01
```

The following shows the distribution of coloured tokens in the SDL channel defined by "**system inres / channel msap1**". Each non-zero number of tokens represents a coloured token determined by a key given in the *state measure*. For instance, the colour **ak1** is represented by 3 colourless tokens and the colour **cc** by 1 colourless token. This measure represents the distribution of signals that could be sent over the channel **msap1** in the system **inres**.

```
State Measure 'Distribution of signals on Channel place
 main.sinres.CHANNELPLACE!msap1!1 server (1=cc,2=ak0,3=ak1,4=dr)'
  distribution
                  0    8.4955042529e-01
                  1    5.1740514888e-02
                  2    3.2107691616e-02
                  3    1.4860853320e-02
                  4    5.1740514888e-02
```

The probability of various signals being at the head of the signal queue of "**system inres / block responder / process responder**" is shown next. Each number represents a different signal, for example, 64 represents the signal **IDISreq**.

```
State Measure 'Mean tokens on Save place
 main.sinres.bresponder.presponder.SAVEPLACE position 0
```

```
(63=iconresp,64=idisreq,65=cr,66=dt1,67=dt0)'
 distribution
                     0      9.6253412170e-01
                    63      7.7610849943e-03
                    64      7.7610849943e-03
                    65      1.3293039762e-02
                    66      2.9721706640e-03
                    67      5.6784978864e-03
```

The probability of the Coxian server modelling the timer in "**system inres / block initiator / process initiator / timer t**" being in a specific phase is given by:

```
State Measure 'Distribution of phases in Cox Timer place
 main.sinres.binitiator.pinitiator.TIMERPLACE!t'
  distribution
                     0      8.0778171122e-01
                     1      5.1086107080e-02
                     2      4.9470983675e-02
                     3      4.7132292263e-02
                     4      4.4528905759e-02
```

Lastly, the throughput of two transitions are given. The first is the throughput of the input transition for "**system inres / block responder / process responder / state disconnected / input cr**". The second is the throughput of the input transition for "**system inres / block responder / process responder / state disconnected / input idisreq**", in this case the result is 0 because the transition never fires.

```
Count Measure 'Throughput for transition
 main.sinres.bresponder.presponder.disconnected+cr'
  mean                  2.5870257444e-03

Count Measure 'Throughput for transition
 main.sinres.bresponder.presponder.disconnected+idisreq'
  mean                  0.0000000000e+00
```

6.5 Mod2sdl

As could be seen in the previous section, the performance results generated for the SDL-net can be quite difficult to understand. The tool **mod2sdl** is a parser written to analyse the results produced by DNAmaca and to format them so that they are understandable to someone who does not have a detailed knowledge of SDL-nets.

The user may specify which performance measures are required, these can include the following:

- the probability of a process being in a Concurrent Communicating System state,
- the information about the process input queue,
- the channel timing information,
- the throughput of transitions and
- the process timer information.

For example, the probability of being in a particular Concurrent Communicating System *state* is given by:

```
Probability of being in:
SYSTEM inres
  BLOCK responder
    PROCESS responder
      STATE connected0                 1.4600031325e-01
      STATE connected1                 5.5494442859e-01
      STATE disconnected               2.7246787734e-01
      STATE wait                       2.6587380822e-02
    PROCESS userresponder
      STATE connec                     2.8799004733e-01
      STATE received0                  5.6514599317e-01
      STATE received1                  1.4686395950e-01
```

Information about the *signal queue* of a process is given by:

```
Probability of Signal in Save queue:
SYSTEM inres
  BLOCK responder
    PROCESS responder
    POSITION 0
      SIGNAL NONE                      9.6253412170e-01
      SIGNAL cr                        1.3293039762e-02
      SIGNAL dt0                       5.6784978864e-03
      SIGNAL dt1                       2.9721706640e-03
      SIGNAL iconresp                  7.7610849943e-03
      SIGNAL idisreq                   7.7610849943e-03
    POSITION 1
      SIGNAL NONE                      9.9999568972e-01
      SIGNAL cr                        4.3102765585e-06
```

Information regarding the throughput of an *input transition* in a process is displayed as:

```
Throughput of input transition in:
SYSTEM inres
  BLOCK initiator
  BLOCK responder
    PROCESS responder
      STATE disconnected
        INPUT cr                          2.5870257444e-03
        INPUT dt0                         0.0000000000e+00
        INPUT dt1                         0.0000000000e+00
        INPUT iconresp                    0.0000000000e+00
        INPUT idisreq                     0.0000000000e+00
      STATE wait
        INPUT cr                          0.0000000000e+00
        INPUT dt0                         0.0000000000e+00
        INPUT dt1                         0.0000000000e+00
        INPUT iconresp                    2.5870257444e-03
        INPUT idisreq                     1.8439830784e-03
```

6.6 Conclusion

The size of the models used was relatively small compared to many real-time systems in use today. The structurally largest SDL system analysed was an automatically generated system containing 1000 SDL states with only one signal passed between the states. One of the most complex system analysed was the protocol shown in the next chapter which contained 5 processes. Another system was a Private Automatic Branch Exchange (PABX) based on an industrial system designed by the author. This included dynamic process creation and deletion.

The purpose of this thesis was to provide an analysis principle that could be used to analyse the performance of systems. The emphasis was not on *efficient* analysis techniques as there is a lot of other work being done in this field [Kno96]. Improvements in these techniques will then allow the technique presented in this thesis to be applied to bigger systems.

The next chapter will elaborate on the application of **DNAsty** to modelling a Concurrent Communicating System implementing InRes [BHS91].

This page intentionally left blank.

Chapter 7

Application of SDL-net

This chapter demonstrates the use of the **DNAsty** toolset by applying it to the **InRes** protocol [BHS91]. A detailed description of the InRes protocol SDL is given on the following pages in graphical SDL, along with an explanation of how the various components of the protocol interrelate. It is shown how a deadlock may occur in the InRes protocol when the *save construct* is removed. A further example demonstrates how the order of signals can affect deadlock, and finally, how performance analysis is done using a variation of performance values.

7.1 InRes Protocol

The InRes protocol [BHS91] modelled in this example is a variation of the *alternating bit protocol* and consists of an **In**itiator entity and a **Res**ponder entity. This protocol is widely used in examples because it contains all the general attributes of a full duplex, sliding window protocol. The Initiator entity starts a data transfer and the Responder entity accepts data from the Initiator. These processes communicate via a Medium process which may lose or change messages. For the performance study a *User* process is added to represent the environment on both the *Initiator* and *Responder* sides. In addition, new signals have been introduced to model the signal sequence number. In the rest of this section, the InRes protocol will be described in more detail using SDL/GR diagrams edited and produced using **SDLite**. A full description of InRes in the SDL/PR format can be found in Appendix B.

The SDL system description of the communicating system using the **InRes** protocol is illustrated in Figure 7.1 and is at the highest level of the SDL hierarchy. The InRes system contains three blocks, a definition of the signals for communication between these blocks and the channels between the blocks.

The **Initiator** block illustrated in Figure 7.2 includes definitions for signals used to communicate between the Initiator process and its *User*. All the communication at this level of the system is done via *signal routes*.

Upon receiving an **ICONreq** signal, the **Initiator** process, illustrated in Figure 7.3, attempts to set up a connection with the Responder by sending a **CR** signal which is acknowledged when a **CC** signal is received. The Initiator indicates to its *User* that a connection is established, this is done by an **ICONconf** signal being returned. For simplicity, only one attempt at setting up a connection is done by the Initiator whereas the protocol usually has 3 retries. Once a connection has been established, the Initiator may receive **IDATreq** data signals from its *User*; these are then transferred to the Responder after attaching a serial number, represented by two different data packets, **DT0** and **DT1**. If a correct acknowledgement to the data packet is received before the timer expires, then the Initiator becomes available to send another data packet.

The *User* process of the Initiator, illustrated in Figure 7.4, establishes a connection and then immediately sends two data signals. The *User* process also initiates a new connection whenever a *disconnect* request is received from the Initiator.

The **Responder** block, illustrated in Figure 7.5, includes definitions for signals used to communicate between the Responder process and its *User*. All the communication at this level of the system is done via *signal routes*.

The **Responder** process, illustrated in Figure 7.6, responds to signals sent to it by the Initiator. The response depends on the state that the Responder is in and on which signals are received from its *User*. The Responder might, for example, be in state **disconnected** and receive signal **CR**, causing it to send an **ICONind** signal to its *User*. If the *User* accepts the connection, then the Responder will receive an **ICONresp** signal, otherwise it will receive an **IDISreq** signal. The response to the Initiator will depend on which of these signals is received from the Responder *User*. If the Responder receives signal **DT0** while it is in state **disconnected**, the signal will then be discarded and no signal will be returned to the Initiator.

The *User* process for the Responder, illustrated in Figure 7.7, only accepts connection requests when it is in state **Connec**. Once the connection is established, it waits until it has received two data signals, after which it requests a disconnection. Any data signals that are received while the connection is being established are discarded.

The **Medium** block, illustrated in Figure 7.8, contains a process that acts as an unreliable medium through which the Initiator and Responder processes have to communicate. The **Medium** process, illustrated in Figure 7.9, simply retransmits the *call establishment* signals, the *disconnect request* signals and the *acknowledgement* signals. It may, however, transmit data packets incorrectly according to a non-deterministic decision (*any*). Specifically, it may lose **DT0** signals or it may incorrectly transmit **DT1** signals.

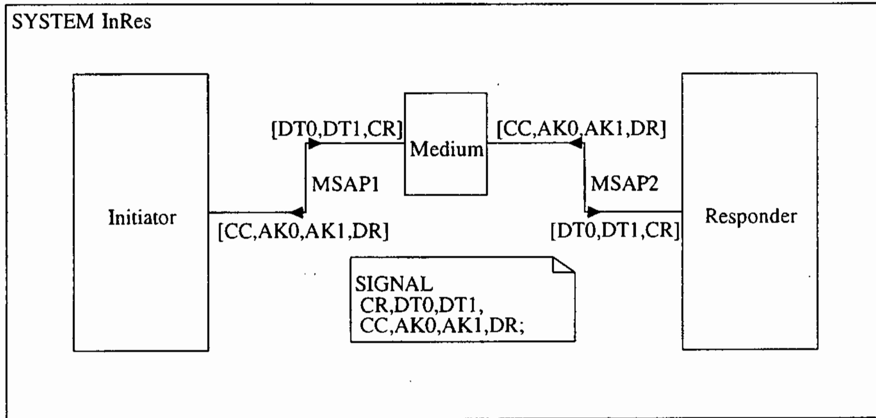

Figure 7.1: InRes system level view

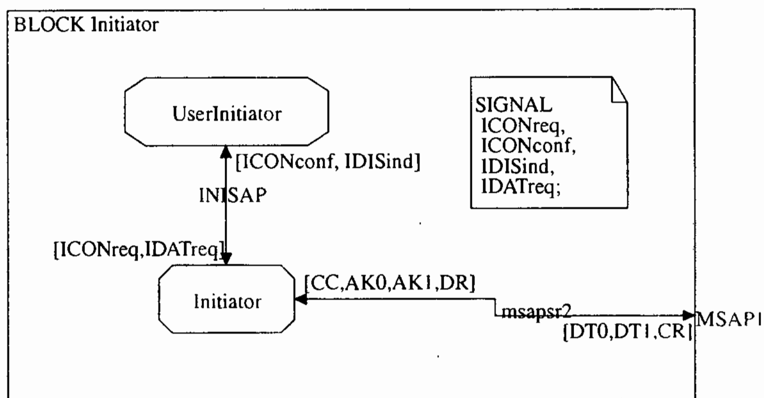

Figure 7.2: InRes Initiator block level view

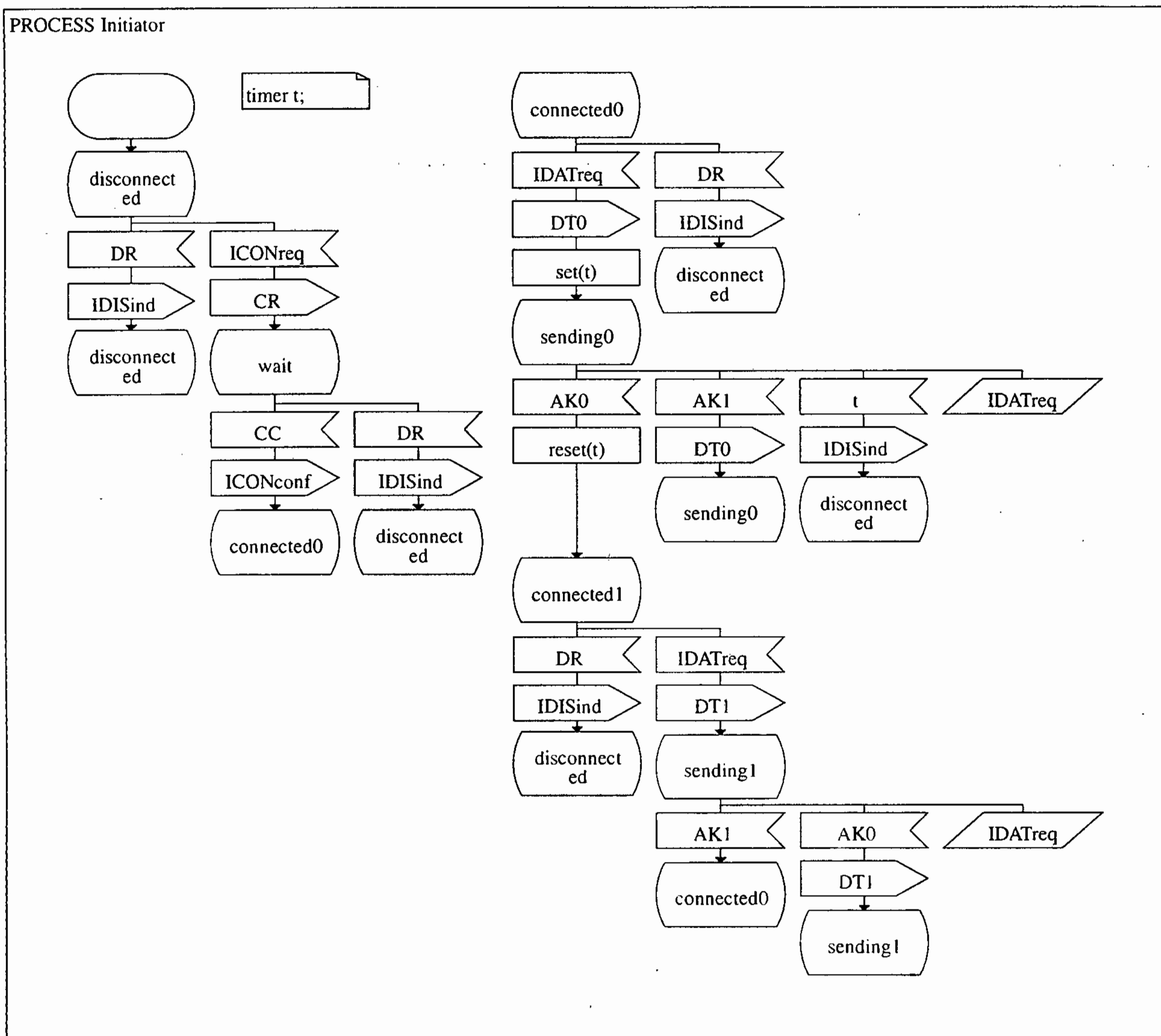

Figure 7.3: InRes Initiator process level view

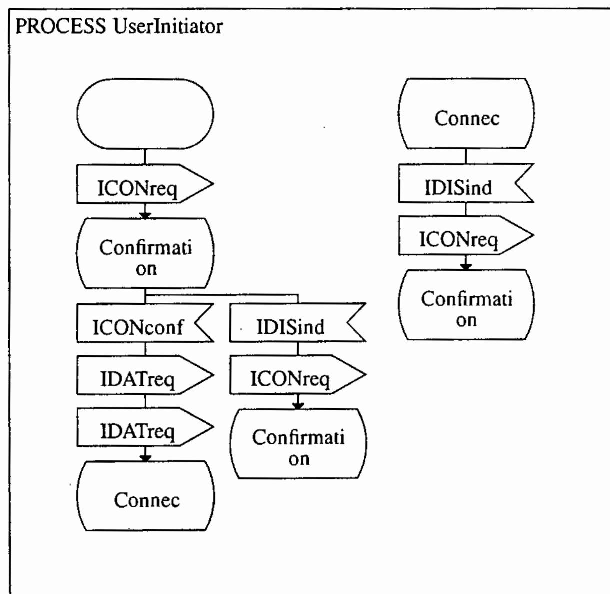

Figure 7.4: InRes User Initiator process level view

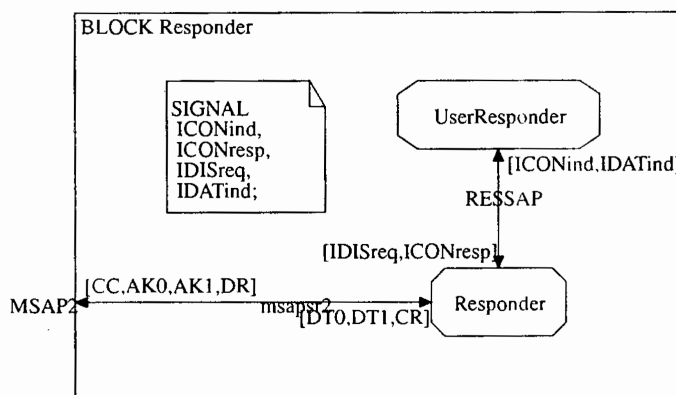

Figure 7.5: InRes Responder block level view

Figure 7.6: InRes Responder process level view

Figure 7.7: InRes User Responder process level view

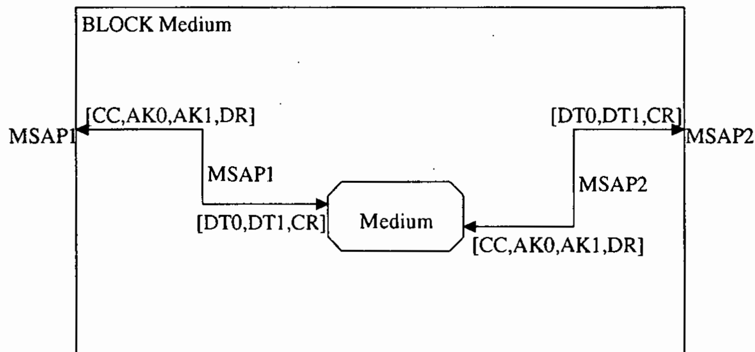

Figure 7.8: InRes Medium block level view

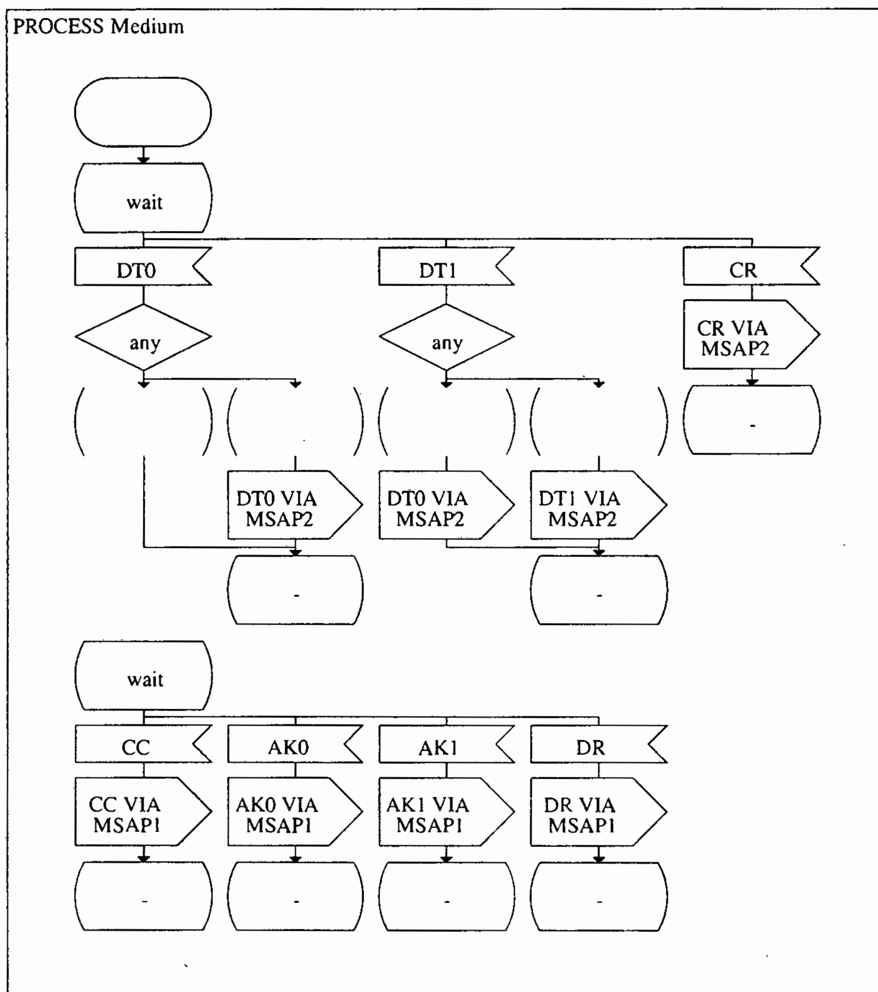

Figure 7.9: InRes Medium process level view

7.2 Correctness

This section describes how **DNAsty** was used to test the correctness of the InRes protocol. It was found that important data signals could be lost and the system could result in a deadlock if the *save* construct was removed from the Initiator process. This deadlock was confirmed with the commercial toolset *Object*GEODE [Ver96b]. In addition, an example is shown that demonstrates the relationship between the SDL-net and its underlying CGSPN with regards to correctness.

It must be remembered that the state space of any real system can become so big that it is no longer possible to analyse it using current techniques and resources. Until this problem is solved, any modelling and analysis technique of real systems will suffer the same problems of having a very big state space as encountered in this thesis. Analytical techniques such as invariant analysis are also not general enough to solve real-world systems.

7.2.1 Deadlock

The InRes protocol was tested for correctness with the **DNAsty** toolset by calculating the place invariants. It was found that the SDL-net representing the InRes protocol was not covered by place invariants. This result was inconclusive as a net not covered by place invariants can still be bounded. The next step was to look at the reachability tree of the model which revealed that the model was infact bounded and would not deadlock. The protocol was then slightly changed by removing the *save* construct from the process Initiator, previously illustrated in Figure 7.3. The *save* construct in the Initiator saves any **IDATreq** signals that arrive at the process while it is in the state **sending0** or **sending1**.

The result was that a data signal (**IDATreq**) was received by the Initiator while it was waiting in state **sending0** for an acknowledgement (**AK0**) from the Responder. Since the data signal was not saved, it was discarded. The Initiator eventually received the acknowledgement and entered the state **connected1** where it was waiting for the data signal from its user. However, the user had already sent this and was waiting for the Initiator to disconnect. This resulted in a deadlock in the protocol. **DNAsty** was used to determine the states that the processes were in when the protocol deadlocked; these states are shown in Table 7.1. In addition, **DNAsty** determined that the timer **t** in process Initiator was inactive.

The same deadlock was found by simulating the InRes protocol with the commercial SDL toolset *Object*GEODE [Ver96b]. The Message Sequence Chart (MSC) [IT93c] in Figure 7.10 shows the order in which signals were sent between the various processes and how this order resulted in the deadlock. This MSC was also generated using the SDL toolset *Object*GEODE.

Finding the deadlock in this way with the **DNAsty** toolset showed that the SDL-net could be used

Process	State
UserInitiator	Connec
Initiator	Connected1
Medium	Wait
Responder	Connected0
UserResponder	Received1

Table 7.1: Process states when deadlock occurred.

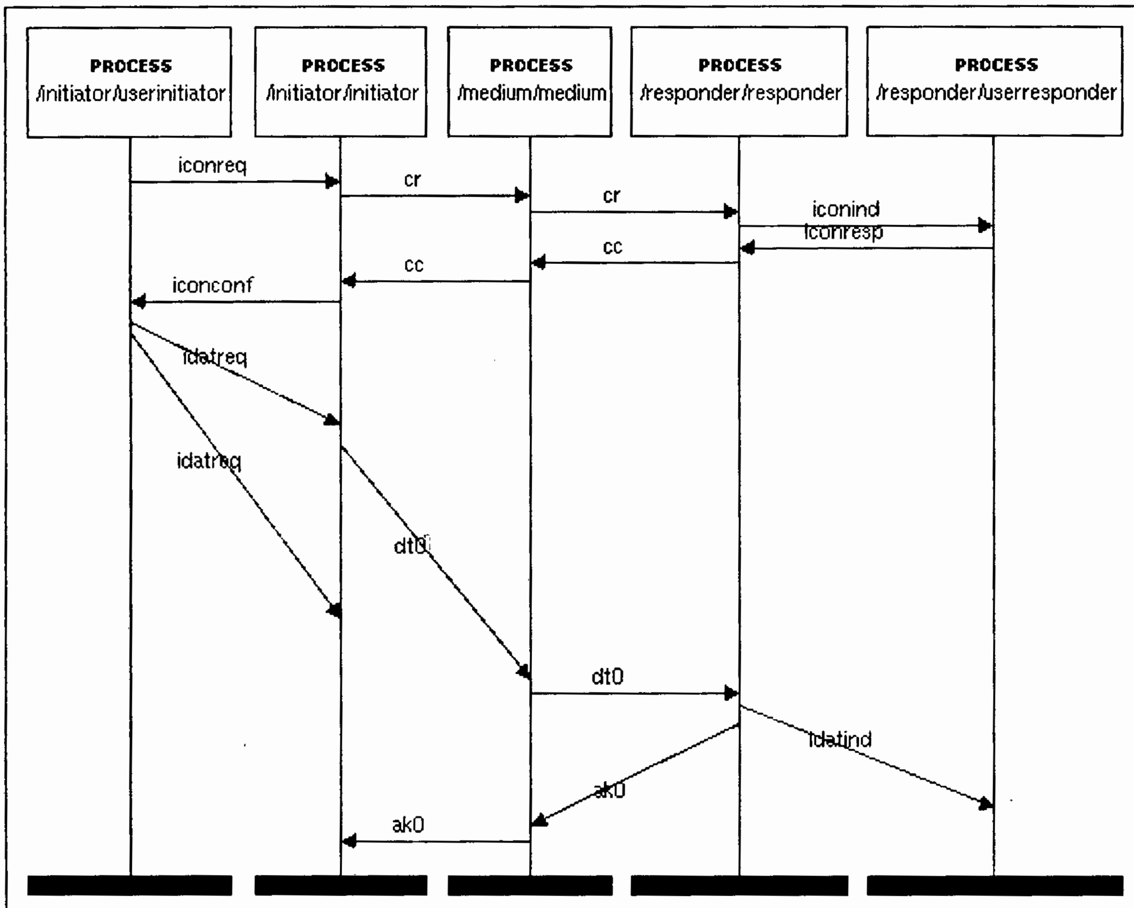

Figure 7.10: Sequence of messages leading to a deadlock.

to find errors in SDL systems. The Save-place reacted in the same way that the *save* construct would be expected to react in SDL.

7.2.2 Order of Signals affecting Correctness

It has been shown that when the underlying CGSPN of a Queueing Petri Net (QPN) is free from deadlocks and is bounded, the QPN is also free from deadlocks and is bounded [Bau93]. This applies also to the SDL-net since it is based on the QPN. Thus one can claim that the SDL-net is free from deadlocks and is bounded if the undelying CGSPN is. Unfortunately the converse is not true, and a SDL-net may still be correct, even if the underlying CGSPN is not correct. By ignoring the order in which the tokens arrive at the special places, transitions might occur that leave the CGSPN in an incorrect state.

As an illustration, consider the SDL-net in Figure 7.11: When the tokens arriving at the **SAVE-PLACE** are queued according to the FIFO scheduling strategy, the order in which the transitions may fire is either **t1→t2→t4→t6→t3** or **t1→t4→t2→t6→t3**. The transition **t5** is not enabled in either of these cases.

However, when the tokens are **not** queued, then the order in which the transitions are fired might be the above mentioned **t1→t2→t4→t6→t3**, **t1→t4→t2→t6→t3**, or it might be **t1→t2→t5** resulting in some **ERRORSTATE**.

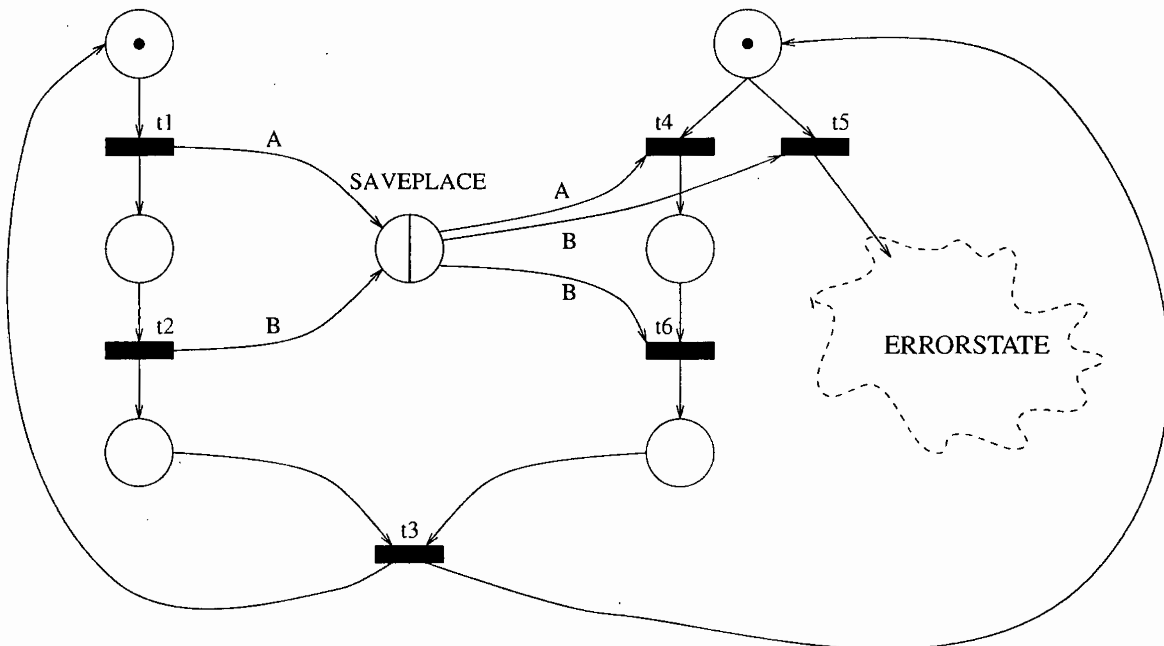

Figure 7.11: SDL-net demonstrating that the order of signals may affect the correctness of the system.

Fortunately, if the correctness analysis done by **DNAsty** reveals that the underlying CGSPN is

correct, then no further correctness analysis needs to be done on the SDL-net. However, if the CGSPN is not correct, a state space exploration technique is then used to analyse the correctness of the SDL-net by considering the order of signals in the special places.

7.3 Performance

With a complex system such as the InRes protocol, it is not always clear what *timeout values* should be used to achieve optimum throughput for a given implementation environment. This section will show how **DNAsty** is used to calculate the *average file transfer* time for a variety of machine speeds and timeout values. It will also examine in which states the Initiator process spends most time; this information can help to determine "bottlenecks" in the protocol.

7.3.1 Experiment description

The performance analysis that was conducted on the InRes protocol is divided into two experiments; the first was an analysis for a variety of channel delays where the SDL action delay (or process speed) was constant, and, the second was an analysis, where the SDL action delay was varied and the channel delay was kept constant. For both these experiments, the timer was varied from 80 to 320 time units in steps of 80.

Experiment 1: Variable Channel Delay

In **experiment 1**, the mean channel delay was **0(40)160** meaning it varied from 0 to 160 time units in steps of 40. Each SDL action was set to take 4 time units and the delay of the timer in the process Initiator was **80(80)320**.

The combination of each of these performance values resulted in 20 individual experiments for which Markov chains were derived. These were solved and the performance results were then applied to the original SDL system.

Experiment 2: Variable SDL Action Delay

In **experiment 2**, the mean delay of each SDL action was *1(1)7*. The mean channel delay was set to 80 time units. As in **experiment 1**, the delay of the timer in the process Initiator was **80(80)320**.

The combination of each of these performance values resulted in 28 individual experiments for which Markov chains were derived. These were also solved and the performance results were then likewise applied to the original SDL system.

7.3.2 Results

It was chosen to investigate the time spent by the Initiator process waiting in various states for signals to arrive. This would give an indication of where the most time is spent in the system. Throughout these discussions, **ChannelDelay** will represent the channel delay, **ActionDelay** the SDL action delay and **TimerDelay** the duration of the timer in process Initiator.

Process Initiator in State *wait*

The proportion of time that process Initiator, described in Figure 7.3, spent in state **wait** during **experiment 1** and **experiment 2** is illustrated in Figure 7.12 and Figure 7.13 respectively.

As is seen from these figures, a great proportion of time is spent in this state. This ranges in Figure 7.12 from about 25%, when **ChannelDelay** = 0 and **TimerDelay** = 320, to over 70%, when **ChannelDelay** = 160 and **TimerDelay** = 80. In Figure 7.13 this ranges from about 33%, when **ActionDelay** = 1 and **TimerDelay** = 320, to over 60%, when **ActionDelay** = 7 and **TimerDelay** = 80.

Both of the figures for the experiments are very similar, as both graphs increase monotonically. The proportion of time spent in state **wait** changes faster in Figure 7.12 than in Figure 7.13, this indicates that the speed of the channel has a greater influence on the proportion of time spent in state *wait* than the speed at which SDL actions are executed.

Process Initiator in State *connected0*

The time spent in state **connected0** is either spent waiting for an **IDATreq** signal from the *User* or waiting for a **DR** signal from the Responder process. Since the communication between the *User* and the Initiator process is via a *signal route* and thus instantaneous, the time before an **IDATreq** signal arrives is influenced only by the **ActionDelay** and not by the **ChannelDelay**.

The proportion of time that process Initiator, described in Figure 7.3, spent in state **connected0** during **experiment 1** and **experiment 2** is illustrated in Figure 7.14 and Figure 7.15 respectively.

In **experiment 1** the proportion of time decreases because the time spent in this state is only partly influenced by **ChannelDelay**. The rate at which the proportion of time decreases is different for each value of **TimerDelay**, with **TimerDelay** = 80 decreasing the fastest and **TimerDelay** = 320 decreasing the slowest.

Experiment 2 showed how the proportion of time spent in the state is influenced by the value of **ActionDelay**, that is, the process speed. Note that the proportion of time spent in this state increases at a faster rate for **TimerDelay** = 80 than for the other values of **TimerDelay** when the value of **ActionDelay** ≤ 4, but it increases at a slower rate when **ActionDelay** > 4.

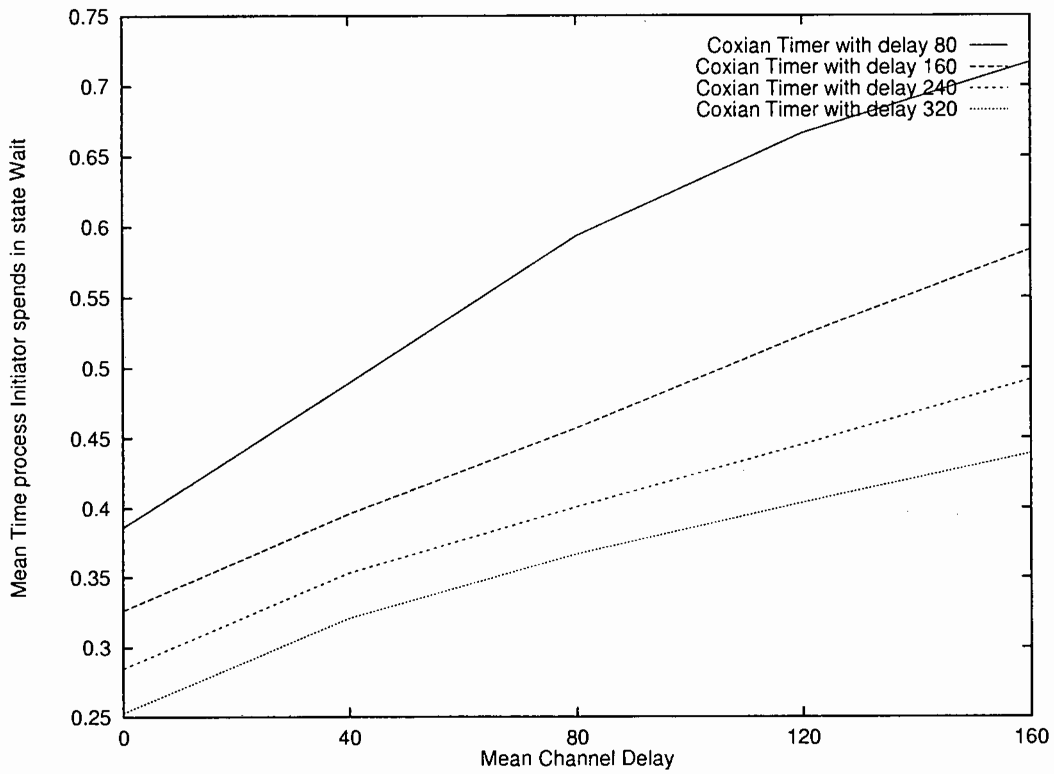

Figure 7.12: Mean time Initiator process spends during **experiment 1** waiting for a Connection Confirm (CC) signal.

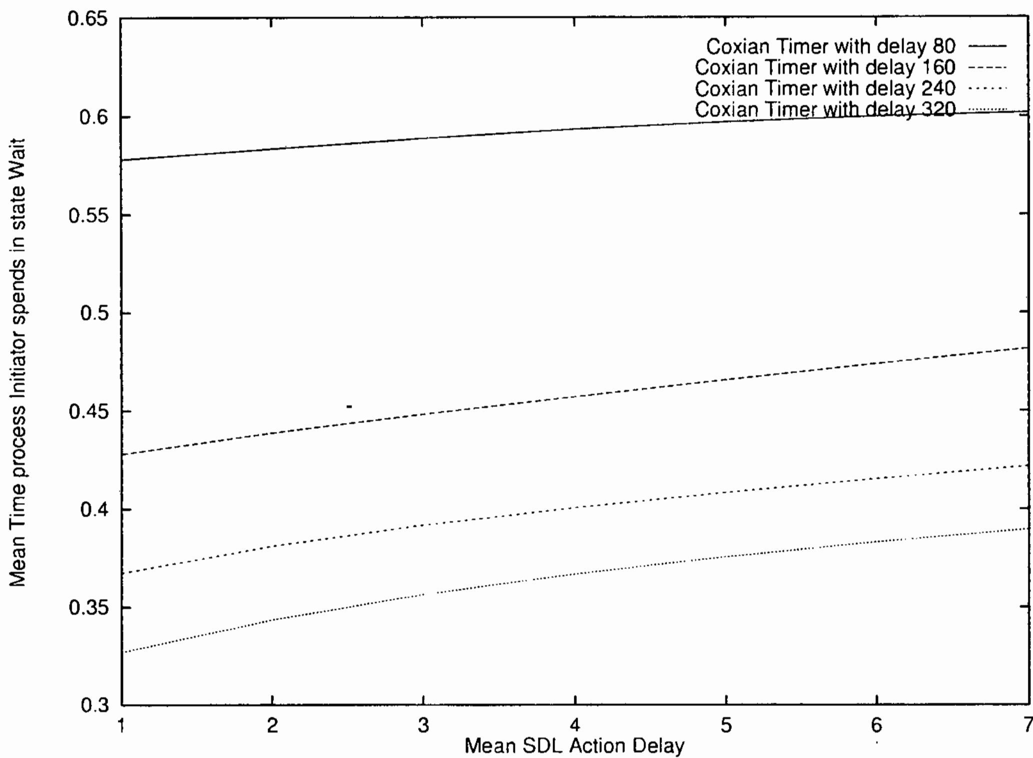

Figure 7.13: Mean time Initiator process spends during **experiment 2** waiting for a Connection Confirm (CC) signal.

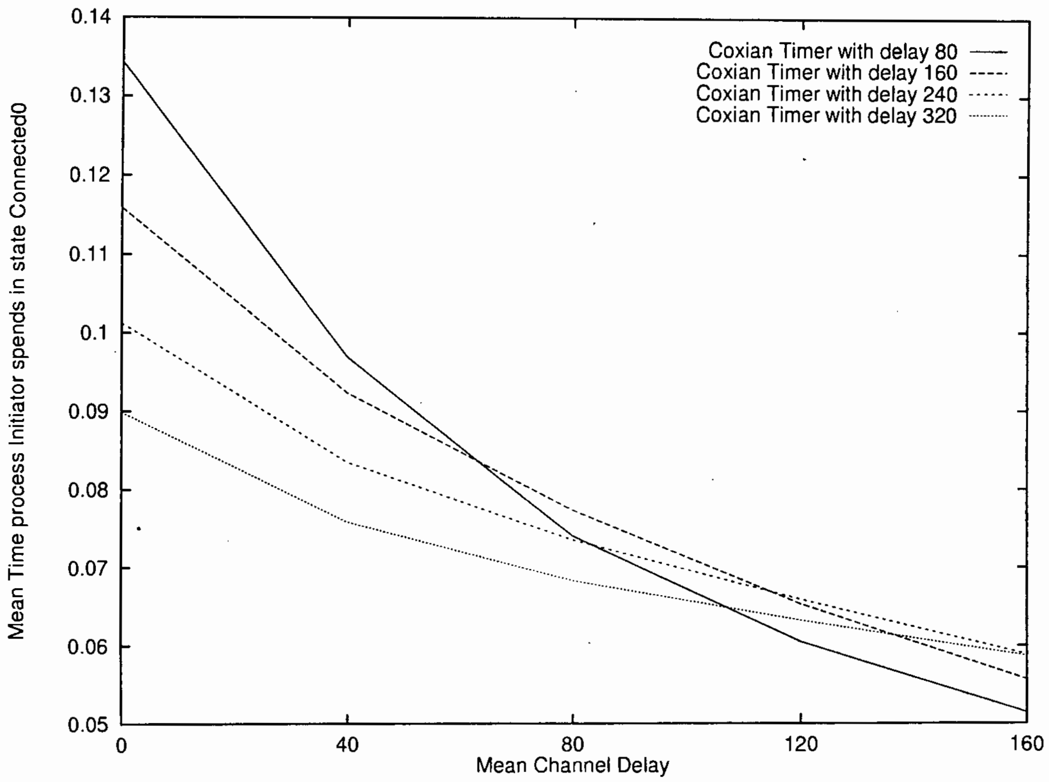

Figure 7.14: Mean time Initiator process spends during **experiment 1** waiting for a Data (IDATreq) signal or a Disconnect (DR) signal.

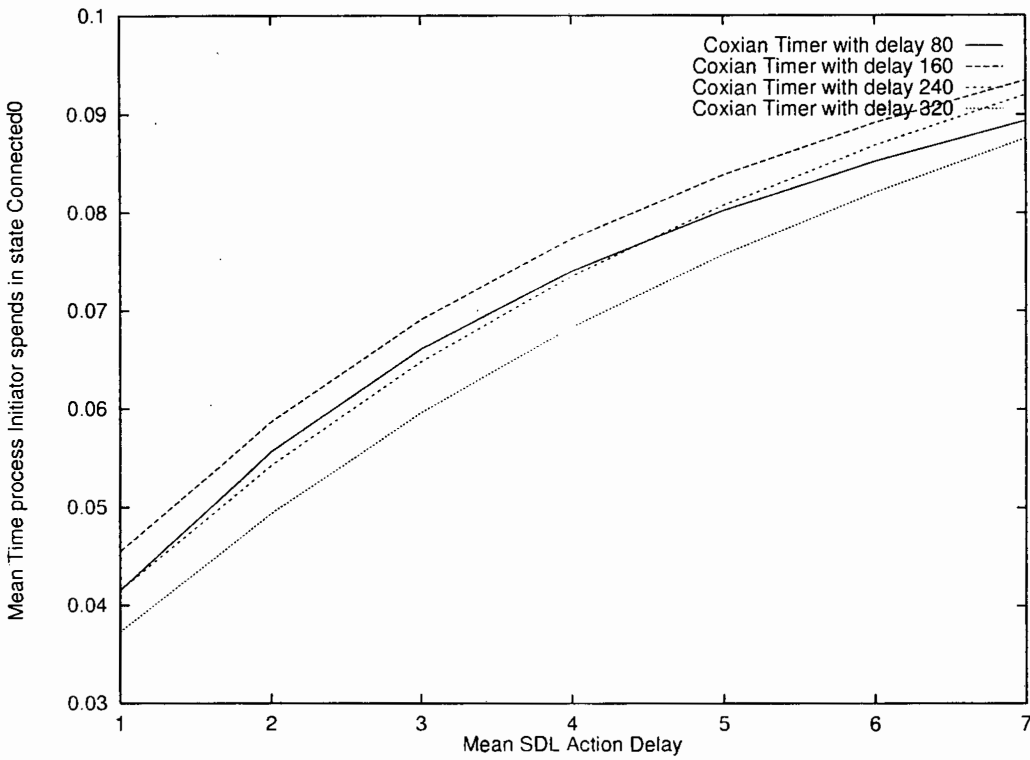

Figure 7.15: Mean time Initiator process spends during **experiment 2** waiting for a Data (IDATreq) signal or a Disconnect (DR) signal.

Process Initiator in State *connected1*

The process only reaches this state if the first data signal **DT0** is not lost, and if the timer does not expire before the responder can transmit an acknowledgement for a successful transmission.

The proportion of time that process Initiator, described in Figure 7.3, spent in state **connected1** during **experiment 1** and **experiment 2** is illustrated in Figure 7.16 and Figure 7.17 respectively.

These curves are similar to those shown for state **connected0**, except that their changes are more extreme. They reveal that significantly less time is spent in state **connected1** than in state **connected0**, sometimes only $\frac{1}{20}$ of the time. This may be because only 2 data signals are sent, so the **DR** signal is only received in state **connected0** and not in state **connected1**. At state **connected1**, only the **IDATreq** signal is thus received, which, as discovered previously, is mainly influenced by the value of **ActionDelay**.

Process Initiator in State *sending1*

Time is spent in state *sending1* waiting for an acknowledgement of the data signal sent to the Responder process. If the signal is transmitted incorrectly via the Medium process, then it will have to be re-transmitted which will increase the time spent in state **sending1**.

The proportion of time that process Initiator, described in Figure 7.3, spent in state **sending1** during **experiment 1** and **experiment 2** is illustrated in Figure 7.18 and Figure 7.19 respectively.

For **experiment 1**, the proportion of time, spent in the state, initially increases with **ChannelDelay**, then becomes constant for about 40 time units and finally decreases again with **ChannelDelay**. The value of **ChannelDelay** where the proportion of time spent in the state is constant, is roughly approximated in **experiment 1** by $\frac{\textbf{TimerDelay}}{2} - 40 \leq \textbf{ChannelDelay} \leq \frac{\textbf{TimerDelay}}{2}$.

In **experiment 2** the proportion of time spent in the state decreases monotonically but at different rates for each value of **TimerDelay**. Note that when **TimerDelay** = 320 the proportion of time spent in the state is influenced only very slightly by the value of **ActionDelay**.

File Transfer Time

The time taken to transfer a file between the *User* of the Initiator process and the *User* of the Responder process during **experiment 1** and **experiment 2** is illustrated in Figure 7.20 and Figure 7.21 respectively.

These results may be used to determine the optimum length of **TimerDelay** given the value for **ChannelDelay** and **ActionDelay**. As is seen in Figure 7.20, when **ChannelDelay** = 0, **TimerDelay** = 80 results in the shortest file transfer time. Similarly, when **ChannelDelay** = 160, **TimerDe-**

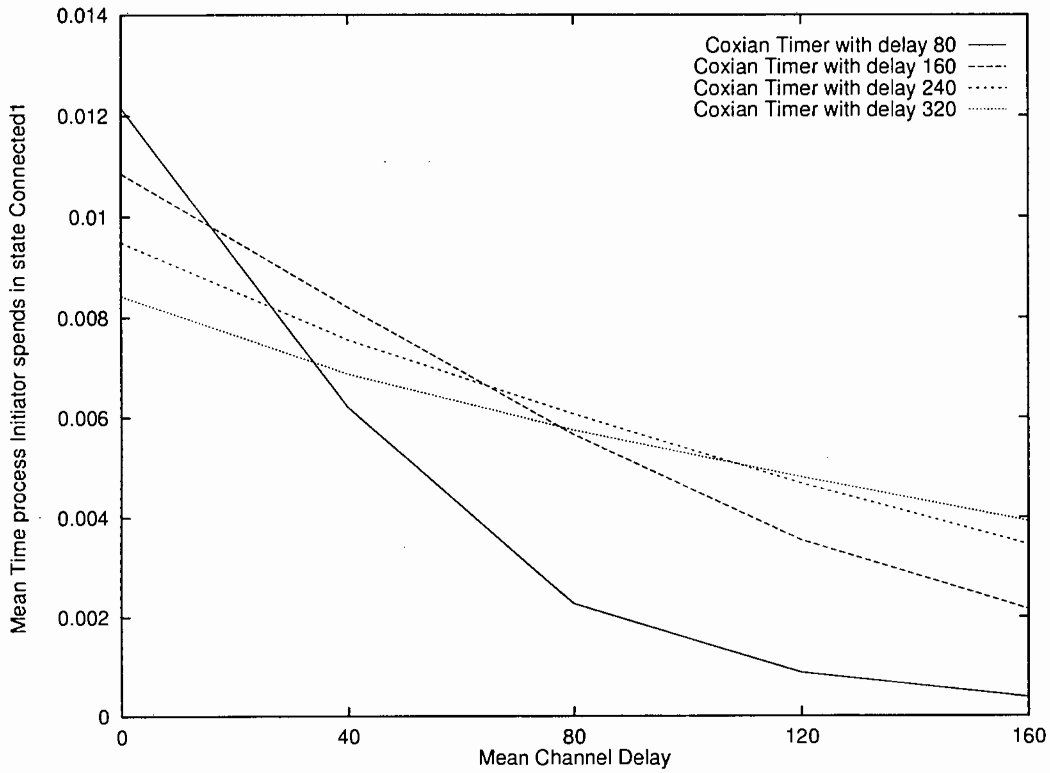

Figure 7.16: Mean time Initiator process spends during **experiment 1** waiting for a Data (IDATreq) signal or a Disconnect (DR) signal.

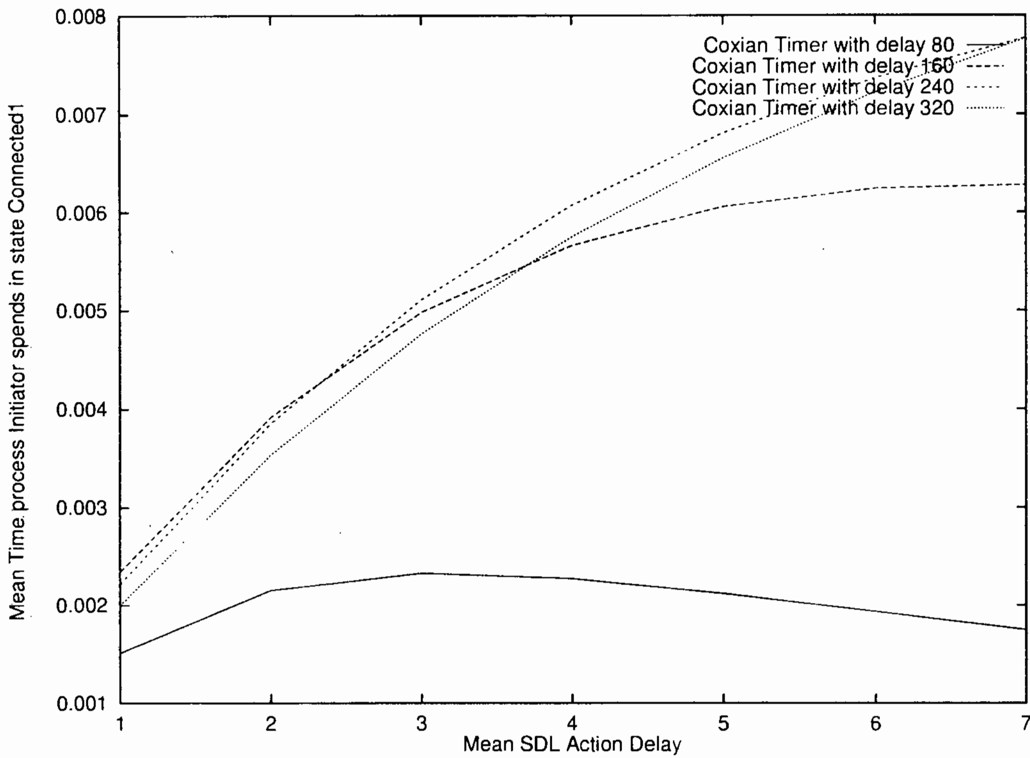

Figure 7.17: Mean time Initiator process spends during **experiment 2** waiting for a Data (IDATreq) signal or a Disconnect (DR) signal.

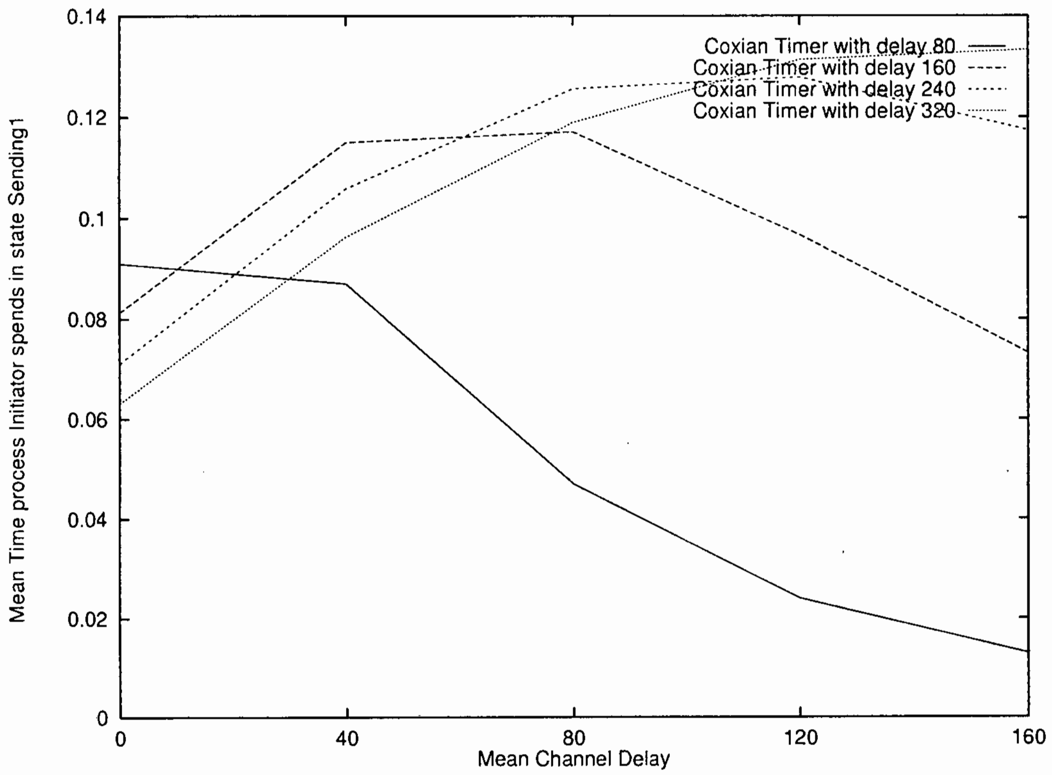

Figure 7.18: Mean time Initiator process spends during **experiment 1** waiting for an Acknowledgement (AK) signal.

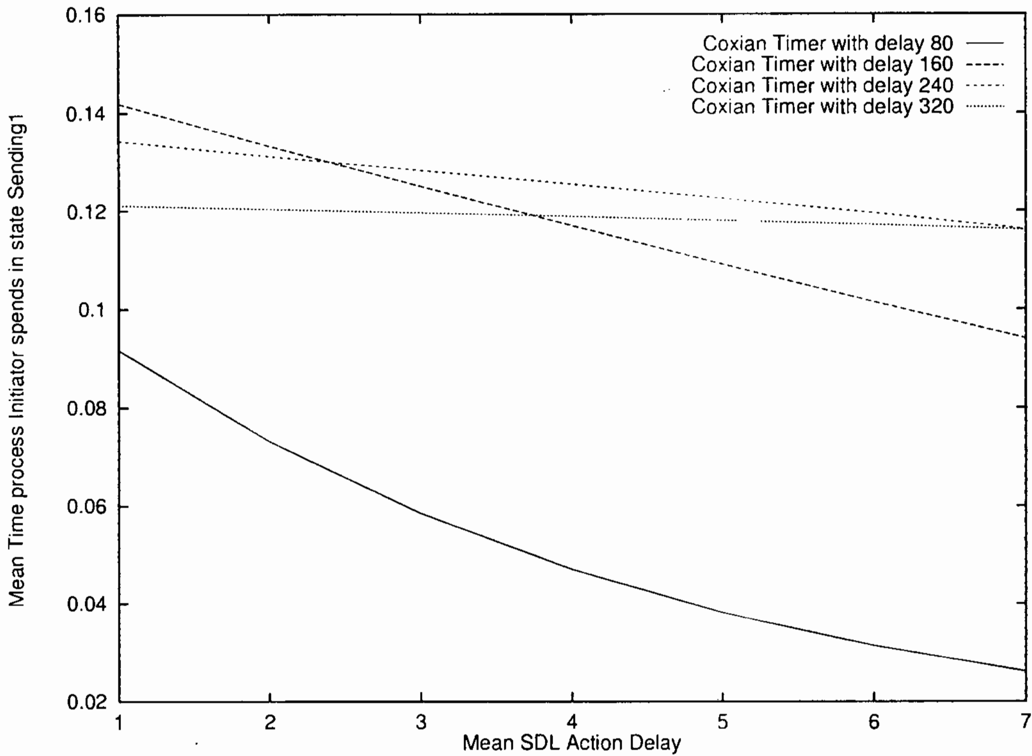

Figure 7.19: Mean time Initiator process spends during **experiment 2** waiting for an Acknowledgement (AK) signal.

lay = 320 results in the shortest file transfer time. In Figure 7.20, the positions at which the graphs for the various values of **TimerDelay** overlap, may be used to determine which timer value to use for a given channel delay

7.3.3 Conclusion

A very common way of evaluating the performance of Concurrent Communicating Systems is to use personal experience in an ad-hoc, subjective manner. The application of the SDL-net in the analysis showed an objective, scientific approach that led to results which would have been difficult to determine using an ad-hoc approach. It would be particularly difficult to predict such results as illustrated in Figure 7.18 by relying on *intuition* and without the aid of a performance analysis toolset such as **DNAsty**.

It would be nice to show the generated SDL-net. However, the automatically generated net is quite large with over 500 arcs, 135 places, 97 timed transitions and 71 immediate transitions. It is not possible to see all the connections without an adequate graphical SDL-net tool which is the reason the InRes SDL-net is not shown in this thesis.

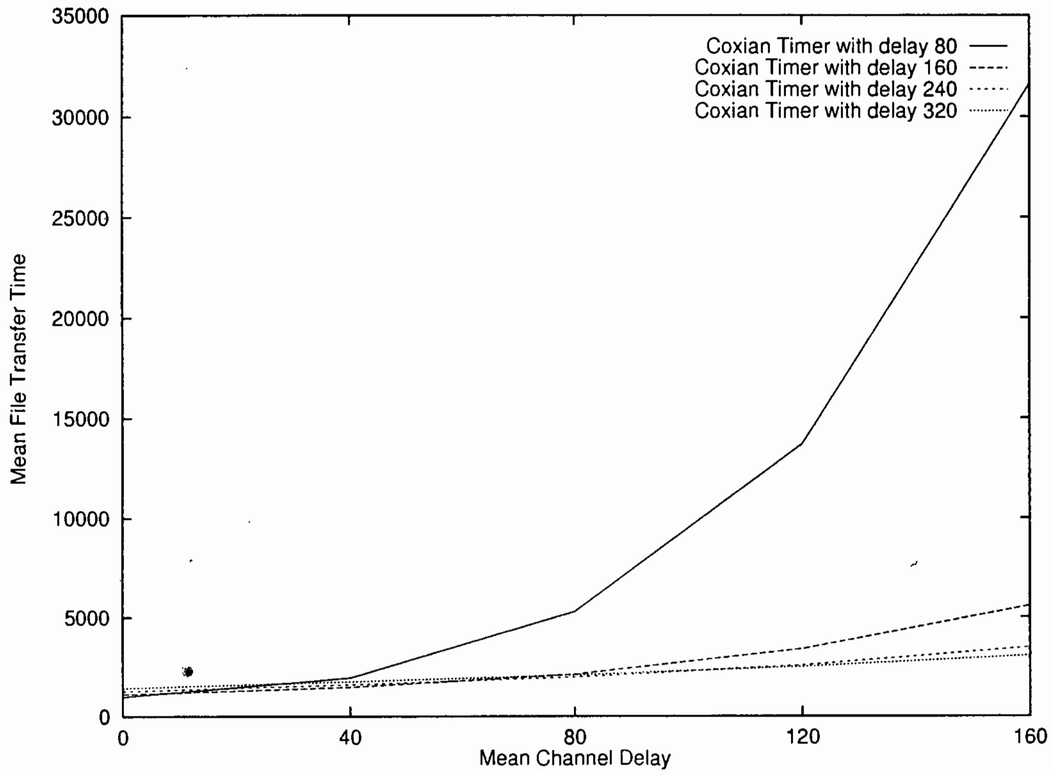

Figure 7.20: Time taken to transfer a file from the Initiator to the Responder during **experiment 1**.

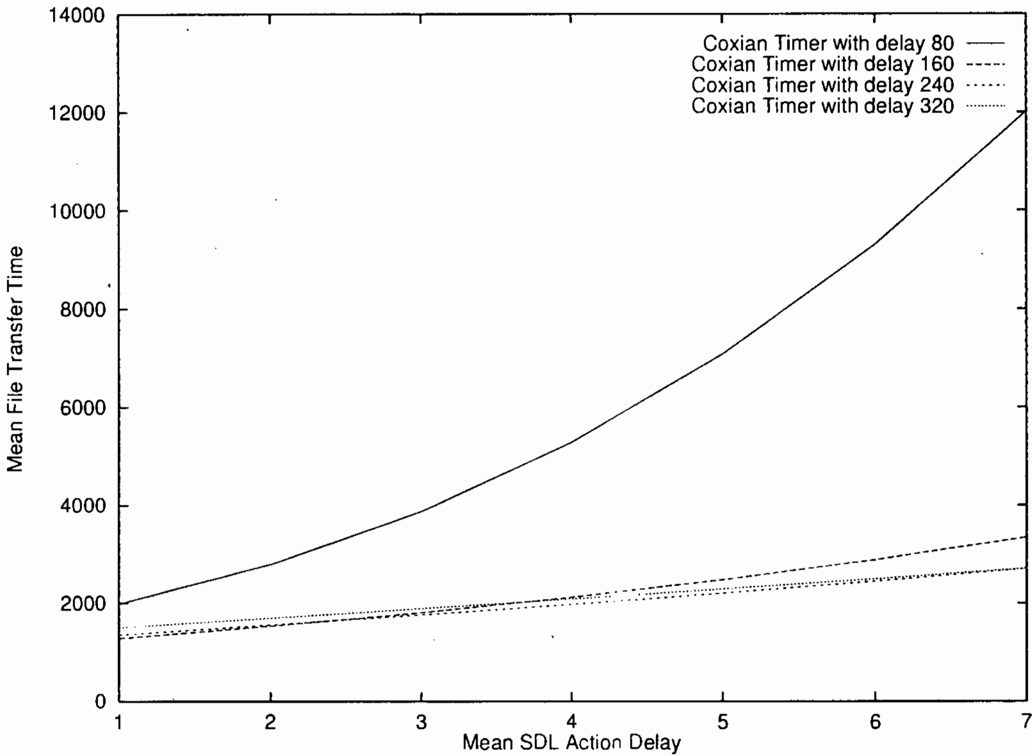

Figure 7.21: Time taken to transfer a file from the Initiator to the Responder during **experiment 2**.

This page intentionally left blank.

Chapter 8

Conclusion

This thesis investigated how a specific stochastic Petri nets could be used to analyse the performance of Concurrent Communicating Systems that have been described with the Specification and Description Language (SDL). An important objective was to make use of existing techniques for analysing stochastic Petri nets. Another objective was that this analysis method should be user-friendly so that the user should not have to be an expert in Petri nets when using this method. In the course of achieving these objectives, the following contributions have been made:

Existing net classes were investigated to determine how they could be used to analyse systems described in SDL.

A new stochastic Petri net, called the SDL-net, was defined to specifically analyse the performance of systems described in SDL. This net introduces new constructs that are used to represent some aspects of SDL.

A mapping from SDL to the SDL-net was defined which required that performance characteristics of SDL be examined and the SDL dynamic semantics be extended where necessary. This was done by augmenting the SDL system with additional external information.

Existing Petri net methods were used to analyse the correctness of a system described as a SDL-net. In addition, the state descriptor of the SDL-net and the transitions between the states were formally defined. These formal definitions were used to derive Markov chains from SDL-nets.

A prototype toolset called **DNAsty** was developed to provide a user-friendly interface to evaluate the performance of systems that can be used even by someone who doesn't know Petri nets. **DNAsty** includes a graphical SDL editor, a tool for mapping the SDL system to a SDL-net, a tool for deriving a Markov chain from a SDL-net and a tool for interpreting the results of solving the Markov chain. This toolset was applied to several SDL systems to analyse their performance, such as a PABX design that included advanced features such as call forwarding.

However, in analysing systems with the SDL-net, the following shortcomings were observed:

It is a problem representing SDL data types in SDL-net. Even though it might be possible to map simple data types such as a bounded integer, such mappings would be impossible in general, without increasing the representive power of the SDL-net.

The state descriptor of the Markov chains derived with this method tended to become quite large. This sometimes made it time-consuming to generate the *state space* for these Markov chains, although solving these Markov chains was relatively quick.

Further research is possible in various related areas based on the results of this thesis.

- The *state space* of the Markov chains did not, in general, depend on the rates and weights of the transitions. Since the generation of the *state space* was the most time-consuming in this analysis, a method could be investigated where the performance measures required by the specified system could be changed, without having to generate the *state space* again.
- A recent development is the hierarchical queueing Petri net (HQPN) [BBK94]. Efficient analysis techniques have been discovered for this net with which very large *state spaces* are analysed. Further research might investigate the definition of a hierarchical SDL-net that uses similar techniques to analyse large systems.
- Since SDL-net is too low-level to represent complex data types, a more high-level net, based on an algebraic Petri net or a predicate/transition net, would perhaps be more suitable to model the data aspects of SDL.
- The use of performance analysis with a specialised net such as the SDL-net could be investigated for other Formal Description Techniques, such as Estelle and LOTOS.
- SDL does not adequately cater for performance characteristics of systems. It would be useful if this could be integrated into the standard so that tool vendors will start using and supporting it.

This thesis demonstrates a rapid, analytical way of analysing performance of Concurrent Communicating System specified in SDL. Even though the state space of such systems is usually quite large, analytical techniques developed for Petri nets could still be utilised to efficiently determine their correctness. This new net provides system designers with important information about how adequately a system will perform, prior to its implementation in software or hardware.

Appendix A

Mathematical definitions of nets

A.1 Nets without time

A.1.1 FIFO queue net (FIFO net)

FIFO nets are Petri nets, where the places are replaced by queues which contain words instead of tokens. The arcs are labeled by words and not integers. This is given more formally [Sou93]:

Definition 25 *A FIFO net is a 4-tuple FN=(P,T,W,A,M_0), where*

- *P is a set of* queues,
- *T is a set of* transitions,
- *W is a* flow function *such that* $W : (P \times T) \cup (T \times P) \rightarrow A^*$,
- *A is a finite* alphabet,
- M_0 *is the* initial marking *of FN, where a marking of FN is a mapping* $M : P \rightarrow A*$.

Definition 26 *A transition t is* enabled *at M, written* $M[t >$ *iff* $\forall$ *queue p we have* $W(p,t) \leq M(p)$ *(where* $\leq$ *stands for the left factor relation). If t is enabled at M, it can occur. Its occurence leads to a reachable marking* M' *(written* $M[t > M'$*).* M' *is obtained from M by removing from each queue p of input of t, the word W(p,t) and by appending to each queue* p' *of output of t, the word W(t,p').*

A.1.2 Predicate-Transition net (Pr/T net)

[Gen86]

Definition 27 *A* Predicate-Transition net *(Pr/T net) is a 4-tuple PRT =* $(PN, \mathcal{L}, \lambda, \mathcal{A})$*, where*

- $PN = (P, T, I^+, I^-, M_0)$ *is the underlying* Place-Transition net,
- *P is a set of* relation symbols, *with given arity,*
- $\mathcal{L}$ *is a* language *disjoint from P,*
- *F is the* flow relation, *where*
  $(p,t) \in F iff I^+(p,t) = 1$ *and* $(t,p) \in F iff I^-(p,t) = 1$
- λ *is a (partial) mapping with domain* $T \cup F$*, inscribing*
  - *some transition t (not necessarily all) with* $\mathcal{L}$*-formulas* φ_t*, called transition-guards,*
  - *each arc* $(x, y) \in F$ *with a finite set* $\lambda(x, y) = \tau_1, \ldots, \tau_m$ *of n-tuples of* $\mathcal{L}$*-terms, where n is the arity of the place belonging to the arc,*
- $\mathcal{A}$ *is a* model *for* $\mathcal{L}$

Extensions to the Basic Pr/T net

[Gen86]

More General Places A considerable gain in flexibility can be achieved if we allow places to be merged or split in small portions. Splitting and merging places requires that any partition of a variable relation into disjoint places should be possible. One of the consequences of this is that the transitions of the Pr/T net may no longer be *uniform* in the sense that all feasible instances of a transition have the same number of state elements in their pre-sets and their post-sets for every predicate.

A.2 Nets with time

A.2.1 Time Petri net (TPN)

[Tau93a]

Definition 28 *A* Time Petri net *is a 3-tuple TPN = (PN,eft,lft), where*

- *PN* = (P, T, I^-, I^+, M_0) *is the underlying* Place-Transition net
- *eft is the* earliest firing time *and lft is the* latest firing time *where* $\forall t \in T$ *eft(t)≤lft(t).*
- *A mapping u:T* $\rightarrow \mathbb{Q}^+ \cup \{-1\}$ *is called a* timer setting *of TPN if* $\forall t \in T$ *with marking M:*
  - *u(t) = -1* $\Leftrightarrow$ *t is not enabled at marking M in PN.*
  - *u(t)* $\neq$ *-1* $\Leftrightarrow$ *t is enabled at marking M in PN and 0* $\leq$ *u(t)* $\leq$ *lft(t).*
- *The* initial state *of the timers of TPN is given by* u_0*, where*

$$u_0(t) := \begin{cases} 0 & if\, t^- \leq M_0 \\ -1 & otherwise \end{cases}$$

  $\forall t \in T$.

The firing of a transition t occurs at the earliest at eft(t) and at the latest at lft(t). It is possible to let eft(t) = lft(t) = 0, which means that the transition t must fire as soon as it is enabled.

Each transition t is equipped with a timer that is set to -1 while t is not enabled, and that keeps the time passed since t has become enabled. All timers are synchronised globally.

A.2.2 Guard Time Petri net (GN)

[Tau93a]

Definition 29 *A* Guard Time Petri net *is a 4-tuple GN = (TPN, W, w,* w_0*), where*

- *TPN = (PN,eft,lft) is a* timed Petri net, *with PN* = (P, T, I^-, I^+, M_0) *being the underlying* Place-Transition net
- *W is the set of* guard *states*
- *the* guard *function w is defined as* $w : W \times T \rightarrow \mathbf{B} \times W$, *where* **B**:=$\{0,1\}$
- $w_0 \in W$ *is the* initial *guard state.*

This page intentionally left blank.

Appendix B

InRes Protocol Described in SDL/PR

This particular SDL description of InRes was adopted from [BHS91] and [Tau93b].

```
SYSTEM InRes;
  SIGNAL CR,DT0,DT1,CC,AK0,AK1,DR;
  CHANNEL MSAP2
    FROM Responder TO Medium WITH CC,AK0,AK1,DR;
    FROM Medium TO Responder WITH DT0,DT1,CR;
  ENDCHANNEL;
  CHANNEL MSAP1
    FROM Medium TO Initiator WITH CC,AK0,AK1,DR;
    FROM Initiator TO Medium WITH DT0,DT1,CR;
  ENDCHANNEL;

  BLOCK Initiator;
    SIGNAL ICONreq,ICONconf,IDISind,IDATreq;
    SIGNALROUTE INISAP
      FROM UserInitiator TO Initiator WITH ICONreq,IDATreq;
      FROM Initiator TO UserInitiator WITH ICONconf, IDISind;
    SIGNALROUTE msapsr2
      FROM ENV TO Initiator WITH CC,AK0,AK1,DR;
      FROM Initiator TO ENV WITH DT0,DT1,CR;
    CONNECT MSAP1 AND msapsr2;

    PROCESS UserInitiator;
      START;
        OUTPUT ICONreq;
        NEXTSTATE Confirmation;
      STATE Confirmation;
        INPUT ICONconf;
          OUTPUT IDATreq;
          OUTPUT IDATreq;
```

```
        NEXTSTATE Connec;
      INPUT IDISind;
        OUTPUT ICONreq;
        NEXTSTATE Confirmation;
    ENDSTATE;
    STATE Connec;
      INPUT IDISind;
        OUTPUT ICONreq;
        NEXTSTATE Confirmation;
    ENDSTATE;
  ENDPROCESS;

  PROCESS Initiator;
    TIMER t;
    START;
      NEXTSTATE disconnected;
    STATE disconnected;
      INPUT DR;
        OUTPUT IDISind;
        NEXTSTATE disconnected;
      INPUT ICONreq;
        OUTPUT CR;
        NEXTSTATE wait;
    ENDSTATE;
    STATE wait;
      INPUT CC;
        OUTPUT ICONconf;
        NEXTSTATE connected0;
      INPUT DR;
        OUTPUT IDISind;
        NEXTSTATE disconnected;
    ENDSTATE;
    STATE connected0;
      INPUT IDATreq;
        OUTPUT DT0;
        SET(t);
        NEXTSTATE sending0;
      INPUT DR;
        OUTPUT IDISind;
        NEXTSTATE disconnected;
    ENDSTATE;
    STATE sending0;
      INPUT AK0;
        RESET(t);
        NEXTSTATE connected1;
```

```
      INPUT AK1;
        OUTPUT DT0;
        NEXTSTATE sending0;
      INPUT t;
        OUTPUT IDISind;
        NEXTSTATE disconnected;
      SAVE IDATreq;
    ENDSTATE;
    STATE connected1;
      INPUT DR;
        OUTPUT IDISind;
        NEXTSTATE disconnected;
      INPUT IDATreq;
        OUTPUT DT1;
        NEXTSTATE sending1;
    ENDSTATE;
    STATE sending1;
      INPUT AK1;
        NEXTSTATE connected0;
      INPUT AK0;
        OUTPUT DT1;
        NEXTSTATE sending1;
      SAVE IDATreq;
    ENDSTATE;
  ENDPROCESS;
ENDBLOCK;

BLOCK Responder;
  SIGNAL ICONind, ICONresp, IDISreq, IDATind;
  SIGNALROUTE RESSAP
    FROM UserResponder TO Responder WITH IDISreq,ICONresp;
    FROM Responder TO UserResponder WITH ICONind,IDATind;
  SIGNALROUTE msapsr2
    FROM ENV TO Responder WITH DT0,DT1,CR;
    FROM Responder TO ENV WITH CC,AK0,AK1,DR;
  CONNECT MSAP2 AND msapsr2;

  PROCESS UserResponder;
    START;
      NEXTSTATE Connec;
    STATE Connec;
      INPUT ICONind;
        OUTPUT ICONresp;
        NEXTSTATE Received0;
    ENDSTATE;
```

```
  STATE Received0;
    INPUT IDATind;
      NEXTSTATE Received1;
  ENDSTATE;
  STATE Received1;
    INPUT IDATind;
      OUTPUT IDISreq;
      NEXTSTATE Connec;
  ENDSTATE;
  STATE *(Connec);
    INPUT ICONind;
      OUTPUT IDISreq;
      NEXTSTATE Connec;
  ENDSTATE;
ENDPROCESS;

PROCESS Responder;
  START;
    NEXTSTATE disconnected;
  STATE disconnected;
    INPUT CR;
      OUTPUT ICONind;
      NEXTSTATE wait;
  ENDSTATE;
  STATE wait;
    INPUT ICONresp;
      OUTPUT CC;
      NEXTSTATE connected1;
  ENDSTATE;
  STATE connected1;
    INPUT DT0;
      OUTPUT IDATind;
      OUTPUT AK0;
      NEXTSTATE connected0;
    INPUT DT1;
      OUTPUT AK1;
      NEXTSTATE connected1;
    INPUT CR;
      OUTPUT ICONind;
      NEXTSTATE wait;
  ENDSTATE;
  STATE connected0;
    INPUT DT1;
      OUTPUT IDATind;
      OUTPUT AK1;
```

```
        NEXTSTATE connected1;
      INPUT DT0;
        OUTPUT AK0;
        NEXTSTATE connected0;
      INPUT CR;
        OUTPUT ICONind;
        NEXTSTATE wait;
    ENDSTATE;
    STATE *;
      INPUT IDISreq;
        OUTPUT DR;
        NEXTSTATE disconnected;
    ENDSTATE;
  ENDPROCESS;
ENDBLOCK;

BLOCK Medium;
  SIGNALROUTE MSAP1
    FROM ENV TO Medium WITH DT0,DT1,CR;
    FROM Medium TO ENV WITH CC,AK0,AK1,DR;
  SIGNALROUTE MSAP2
    FROM ENV TO Medium WITH CC,AK0,AK1,DR;
    FROM Medium TO ENV WITH DT0,DT1,CR;
  CONNECT MSAP1 AND MSAP1;
  CONNECT MSAP2 AND MSAP2;

  PROCESS Medium;
    START;
      NEXTSTATE wait;
    STATE wait;
      INPUT DT0;
        DECISION any;
          () :
          () : OUTPUT DT0 VIA MSAP2;
        ENDDECISION;
        NEXTSTATE -;
      INPUT DT1;
        DECISION any;
          () : OUTPUT DT0 VIA MSAP2;
          () : OUTPUT DT1 VIA MSAP2;
        ENDDECISION;
        NEXTSTATE -;
      INPUT CR;
        OUTPUT CR VIA MSAP2;
        NEXTSTATE -;
```

```
        INPUT CC;
          OUTPUT CC VIA MSAP1;
          NEXTSTATE -;
        INPUT AK0;
          OUTPUT AK0 VIA MSAP1;
          NEXTSTATE -;
        INPUT AK1;
          OUTPUT AK1 VIA MSAP1;
          NEXTSTATE -;
        INPUT DR;
          OUTPUT DR VIA MSAP1;
          NEXTSTATE -;
      ENDSTATE;
    ENDPROCESS;
  ENDBLOCK;
ENDSYSTEM;
```

Bibliography

[ABKK95] A. Attieh, M.C. Brady, W.J. Knottenbelt, and P.S. Kritzinger. Functional and temporal analysis of concurrent systems. In *Proc. of Protocol Performance Workshop, held in conjunction with the 16th International Conference on the Theory and Application of Petri nets, Turin, Italy*, pages 79 – 96. Springer-Verlag, Berlin, 1995.

[Bau93] F. Bause. Queueing Petri Nets: a Formalism for the Combined Qualitative and Quantitative Analysis of Systems, Toulouse (France). In *Proceedings of the 5th International Workshop on Petri Nets and Performance Models*. IEEE, October 1993.

[BB90] F. Bause and P. Buchholz. Protocol Analysis using a timed version of SDL. In *FORTE'90, 3rd Int'l Conf. on Formal Description Techniques*, pages 239–255. North Holland Elsevier, 1990.

[BBK94] F. Bause, P. Buchholz, and P. Kemper. Hierarchically combined Queueing Petri Nets. In *G. Cohen, J. P. Quadrat (eds.), 11th Int. Conference on Analysis and Optimizations of Systems, Springer LNCIS 199*, pages 176–182, 1994.

[BH93] R. Braek and Ø. Haugen. *Engineering real time systems*. Prentice-Hall International, 1993.

[BHS91] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification*. Prentice-Hall International, 1991.

[Bil94] J. Billington. High-Level Net Model of X25 Call Control Procedures with Petri Nets. In J. Billington, editor, *Application of Petri Nets to Communication Protocols. Advanced Practical Tutorial Notes, 15th International Conference on Application and Theory of Petri Nets, Zaragoza, Spain*, June 1994.

[BK91] F. Bause and P. Kemper. *QPN-Tool User's Guide*, 1.0 edition, 1991.

[BK96] F. Bause and P.S. Kritzinger. *Introduction to Stochastic Petri Net Theory*. Vieweg Verlag, Wiesbaden, Germany, 1996.

[BMSK96] M. Bütow, M. Mestern, C. Schapiro, and P.S. Kritzinger. Performance Modelling with the Formal Specification Language SDL. In *Proceedings of the FORTE/PSTV'96 conference on Formal Description Techniques, Kaiserslautern, Germany*. Chapman and Hall, 1996.

[Cox55] D.R. Cox. A use of complex probabilities in the theory of stochastic processes. In *Proc. of Cambridge Phil. Soc.*, volume 51, 1955.

[CS61] D.R. Cox and W.L. Smith. *Queues*. Methuen's Monographs on Statistical Subjects. Spottiswoode, Ballantyne & Co. Ltd., London and Colchester, 1961.

[DH90] C. Dimitrovici and U. Hummert. Composition of algebraic high-level nets. In *Proc. of the 7th ADT-Workshop, Lecture Notes in Computer Science 534 (1991)*. Springer-Verlag, Berlin, 1990.

[DHHMC95] M. Diefenbruch, E. Heck, J. Hintelmann, and B. Müller-Clostermann. Performance Evaluation of SDL Systems Adjunct by Queueing Models. In A. Sarma R. Bræk, editor, *SDL'95 with MSC in CASE*. North Holland Elsevier, 1995.

[FKPP95] F.Bause, H.M. Kabutz, P.Kemper, and P.S.Kritzinger. SDL and Petri net performance analysis of communicating systems. In *15th Int'l Conf. on Protocol Specification, Testing and Verification, Warsaw, Poland*. Chapman and Hall, 1995.

[Gen86] H.J. Genrich. Predicate/Transition Nets. *Advances in Petri Nets*, 1986.

[GP87] E. Gelenbe and G. Pujolle. *Introduction to queueing networks*. John Wiley & Sons Ltd., 1987.

[Gra90] J. Grabowski. Statische und dynamische Analysen für SDL-Spezifikationen auf der Basis von Petri-Netzen und Sequence-Charts. Master's thesis, University of Bern, 1990.

[Hog89] D. Hogrefe. *Estelle, LOTOS und SDL*. Springer-Verlag, Berlin, 1989.

[Hol91] G.J. Holzmann. *Design and validation of Computer Protocols*. Prentice-Hall International, 1991.

[IT93a] ITU-T. *CCITT Specification and Description Language (SDL) - SDL Formal Definition: Dynamic Semantics, Z.100, Annex F.3*. International Telecommunication Union, 1993.

[IT93b] ITU-T. *CCITT Specification and Description Language (SDL), Z.100*. International Telecommunication Union, 1993.

[IT93c] ITU-T. *Message Sequence Chart (MSC), Z.120.* International Telecommunication Union, 1993.

[IT93d] ITU-T. *SDL Methodology Guidelines, Z.100, Appendix I.* International Telecommunication Union, 1993.

[IT94] ITU-T. *First outline document for SDL Methodology, Z.100.* International Telecommunication Union, Oct 1994.

[IT96] ITU-T. *Draft New Recommendation Z.106: CIF for SDL.* International Telecommunication Union, Feb 1996.

[Jen94] K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. In *Proceedings of the 15th International Conference on Application and Theory of Petri Nets.* Springer-Verlag, Berlin, 1994.

[Kab96] H.M. Kabutz. *SDLite v1.0 User Manual.* Data Network Architectures Laboratory, University of Cape Town, Rondebosch, 7700, South Africa, 1996.

[Kan92] K. Kant. *Introduction to computer system performance evaluation.* McGraw-Hill Inc., 1992.

[KBA94] W.J. Knottenbelt, M. Brady, and A. Attieh. *DNAnet User Guide.* Data Network Architectures Laboratory, University of Cape Town, Rondebosch, 7700, South Africa, 1994.

[KCYH91] H.C. Kim, W. Choi, C.H. Yim, and J.P. Hong. The Automated Verification of SDL Specifications Using Numerical Petri-nets. In *SDL '91 - Evolving Methods.* North Holland Elsevier, 1991.

[KMT88] E. Kettunen, E. Montonen, and T. Tuuliniemi. An Interactive PrT-Net Tool for Verification of SDL-Specifications. Technical Report 3, Helsinki University of Technology, Digital Systems Laboratory, Otaniemi, 1988.

[Kno96] W.J. Knottenbelt. Generalised Markovian Analysis of Timed Transition Systems. Master's thesis, University of Cape Town, 1996.

[Leh93] J. Lehnert. Netzdarstellung und Petri-Netz-Analyse von SDL-Spezifikationen. Master's thesis, Fachbereich Informatik, Humboldt Universität zu Berlin, 1993.

[Neu] P. Neumann. Forum On Risks To The Public In Computers And Related Systems. http://catless.ncl.ac.uk/Risks.

[Neu95] P. Neumann. *Computer Related Risks*. Addison-Wesley Publishing Company, 1995.

[OFMP+94] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J.R.W. Smith. *Systems engineering using SDL-92*. North Holland Elsevier, 1994.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Bonn, 1962.

[Rei91] W. Reisig. Petri Nets and Algebraic Specifications. *Theoretical Computer Science*, 80, 1991.

[Rog95] RogueWave. *zApp Programmer's Guide*. Rogue Wave Software, Inc., P.O.Box 2328, Corvallis, OR 97339, USA, 1995.

[SC81] C.H. Sauer and K.M. Chandy. *Computer Systems Performance Modeling*. Prentice-Hall International, 1981.

[Sou93] Y. Souissi. A modular approach for the validation of communication protocols using FIFO nets. Technical report, INT, 1993.

[Ste94] W.J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

[Str91] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, second edition, 1991.

[Tau93a] U. Taubert. Entwicklung eines Analysewerkzeuges für SDL'92 Spezifikationen unter Nutzung erweiterter Petri-Netze. Master's thesis, Fachbereich Informatik, Humboldt Universität zu Berlin, 1993.

[Tau93b] U. Taubert. Vergleich zweier Methoden zur Analyse von SDL-Spezifikationen mit Hilfe von Petri-Netzen am Beispiel des Inres-Protokolls. Technical report, Fachbereich Informatik, Humboldt Universität zu Berlin, 1993. Jahresarbeit.

[TFD94] U. Taubert, J. Fisher, and E. Dimitrov. Analysis and formal Verification of SDL'92 Specifications using Extended Petri Nets. Technical report, Fachbereich Informatik, Humboldt Universität zu Berlin, 1994.

[Ver95] Verilog. *Object***GEODE***: SDL Application Programming Interface (Reference Manual)*. Verilog SA, France, 150 rue Nicolas Vauquelin, 31106 Toulouse cedex, 1995.

[Ver96a] Verilog. *Object***GEODE***: SDL C Code Generator - Reference Manual*. Verilog SA, France, 150 rue Nicolas Vauquelin, 31106 Toulouse cedex, 1996.

[Ver96b] Verilog. *Object***GEODE***: SDL Editor - User's Guide.* Verilog SA, France, 150 rue Nicolas Vauquelin, 31106 Toulouse cedex, 1996.

[Wal88] J. Walrand. *An introduction to queueing networks.* Prentice-Hall International, 1988.

[Whe93] G. Wheeler. *Protocol Engineering from Estelle Specifications.* PhD thesis, University of Cape Town, 1993.

[WK92] G. Wheeler and P.S. Kritzinger. PEW: A Tool for the Automated Performance Analysis of Protocols. *South African Computer Journal*, 8, 1992.