

VERIFICATION OF SDL SYSTEMS WITH PARTIAL ORDER METHODS

A THESIS SUBMITTED TO  
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES  
OF  
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

MEHMET ALPER ŞEN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE  
DEGREE OF  
MASTER OF SCIENCE  
IN  
THE DEPARTMENT OF ELECTRICAL AND ELECTRONICS  
ENGINEERING

MAY 1997

Approval of the Graduate School of Natural and Applied Sciences.

---

Prof. Dr. Tayfur Öztürk  
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

---

Prof. Dr. Fatih Canatan  
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

---

Prof. Dr. Kemal İnan  
Supervisor

Examining Committee Members

Assoc. Prof. Dr. Güney Gönenç (Chairman)

Prof. Dr. Kemal İnan

Prof. Dr. Semih Bilgen

Assoc. Prof. Dr. M. Mete Bulut

Assist. Prof. Dr. Halit Oğuztüzün

# ABSTRACT

VERIFICATION OF SDL SYSTEMS WITH PARTIAL ORDER METHODS

Şen, Mehmet Alper

M.S., Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Kemal İnan

May 1997, 99 pages

Partial order methods are one of the state space reduction techniques for formal verification of industrial size systems. SDL specific complexity relief techniques are derived by applying the method of persistent sets, which takes benefit of partial order methods, to SDL programs. For this purpose an automaton called SSM is defined that models SDL programs by abstracting out linguistic details. Necessary and sufficient characterizations of persistent sets are derived for communicating SSMs and used to formulate an algorithm of persistent set computation. The subset of SDL for which the results are derived covers SDL primitives *save* and *priority inputs* that violate FIFO reading discipline at input queues. The results of the approach have been implemented with a software tool POVSDL and experimental results show that complexity reduction can be orders of magnitude.

Keywords: verification, partial order reduction, persistent set, formal specification, parser, software engineering, communication software, SDL

# ÖZ

## SDL SİSTEMLERİNİN KISMİ SIRALAMA METODLARI İLE DOĞRULANMASI

Şen, Mehmet Alper

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Kemal İnan

Mayıs 1997, 99 sayfa

Kısmi sıralama metodları endüstriyel boyuttaki sistemlerin biçimsel olarak doğrulanmasında kullanılan durum alanı ufaltım tekniklerinden biridir. Kısmi sıralama metodlarından yararlanan sürekli kümeler metodunun SDL programlarına uygulanması ile SDL sistemleri için karışıklıktan kurtaran teknikler türetilmiştir. Bu nedenle SDL programlarını, SDL dilinin detaylarını soyutlayarak, modelleyen SSM adlı bir otomat tanımlanmıştır. Haberleşen SSM'ler için sürekli kümelerin gerekli ve yeterli karakterizasyonu türetilmiştir ve bu da sürekli küme hesaplanmasında bir algoritma formüle edilmesinde kullanılmıştır. Sonuçların türetildiği SDL alt kümesi *save* ve *priority inputs* gibi giriş dizisinde FIFO okuma disiplini ni ihlal eden SDL kavramlarını kapsamaktadır. Geliştirilen yaklaşımın sonuçları POVSDL adlı bir yazılım aracı ile gerçekleştirilmiştir ve deney sonuçları karışıklık azaltımının büyük oranda olduğunu göstermiştir.

Anahtar Kelimeler: doğrulama, kısmi sıralama azaltımı, sürekli küme, biçimsel belirtim, ayrıştırıcı, yazılım mühendisliği, iletişim yazılımı, SDL

## ACKNOWLEDGMENTS

I would like to express my deep gratitude to my thesis supervisor Prof. Dr. Kemal İnan for his guidance both in this study and the various aspects of life.

My special thanks are due to dear friend Hüsnü Yenigün. His collaboration and invaluable discussions, criticism and close cooperation helped to improve the quality of the thesis.

I must thank nice people Cevdet Dengi, Uygur Doyuran and Ali Sezgin whom I met at Software Center for their friendship and answering my endless questions.

Finally I would like to thank my family and my friends for their moral support and encouragement and without whom life would not be that easy after all.

# TABLE OF CONTENTS

ABSTRACT . . . . .	iii
ÖZ . . . . .	iv
ACKNOWLEDGMENTS . . . . .	v
TABLE OF CONTENTS . . . . .	vi
LIST OF TABLES . . . . .	viii
LIST OF FIGURES . . . . .	ix
LIST OF ABBREVIATIONS . . . . .	x
CHAPTER	
I INTRODUCTION . . . . .	1
I.1 Background . . . . .	1
I.2 Partial Order Methods . . . . .	3
I.3 Overview . . . . .	4
II SDL . . . . .	8
III SDL-LIKE STATE MACHINE (SSM) . . . . .	13
III.1 SSM . . . . .	14
III.2 Independency and Traces . . . . .	16
IV REDUCED SSM BY PERSISTENT SETS . . . . .	19
IV.1 Persistent Sets . . . . .	19
IV.2 The Basic Case . . . . .	22
IV.2.1 Reduction theorems . . . . .	22
IV.2.2 Algorithm . . . . .	25
IV.3 Extensions . . . . .	26
IV.3.1 Reduction theorems . . . . .	28

	IV.3.2	Algorithm . . . . .	31
	IV.4	Further SDL Constructs : Variables and Transition Atomicity . . . . .	34
V		EXPERIMENTS . . . . .	37
	V.1	POVSDL tool . . . . .	37
		V.1.1 Scanner and Parser . . . . .	38
		V.1.2 POVSDL software . . . . .	39
	V.2	Experimental results . . . . .	44
VI		CONCLUSION . . . . .	47
	VI.1	Summary . . . . .	47
	VI.2	Future Work . . . . .	48
		REFERENCES . . . . .	49
		APPENDICES . . . . .	52
	A	POVSDL SCANNER SPECIFICATION . . . . .	52
	B	POVSDL PARSER SPECIFICATION . . . . .	55
	C	MAIN.CC SOFTWARE CODE . . . . .	58
	D	QUEUE_INT.H SOFTWARE CODE . . . . .	77
	E	STACK.H SOFTWARE CODE . . . . .	79
	F	ISDN SPECIFIED IN INPUT LANGUAGE OF POVSDL . . . . .	81

# LIST OF TABLES

## TABLE

V.1 Results of verification . . . . .	46
---------------------------------------	----

# LIST OF FIGURES

## FIGURES

II.1 Basic constructs for the description of a process . . . . .	9
IV.1 State space generation with Persistent set . . . . .	20
IV.2 Reachable states generated with two methods . . . . .	23
IV.3 <i>O,I</i> and <i>J</i> type processes . . . . .	29
IV.4 An example SDL system . . . . .	33
V.1 Makefile for the generation of POVSDL . . . . .	40
V.2 Interaction of tools to generate the lexical analyzer, parser and POVSDL . . . . .	41
V.3 Graphical representation of an example SDL system and the cor- responding compiled system for POVSDL tool input . . . . .	43

## LIST OF ABBREVIATIONS

CRA	Conventional Reachability Analysis.
LALR	LALR(1) parser generator.
POVSDL	Partial Order Verifier for SDL.
PRA	Persistent Reachability Analysis.
REX	Lexical analyzer generator.
SDL	Specification and Description Language.
SDL/GR	SDL Graphical Representation.
SDL/PR	SDL Phrase Representation.
SSM	SDL-like State Machine.

# CHAPTER I

## INTRODUCTION

### I.1 Background

Concurrent systems are systems composed of elements that can operate concurrently and communicate with each other. Each component can be viewed as a reactive system, that is a system that continuously interacts with its environment. The behavior of a reactive system is defined by its ongoing behavior over time. This is quite unlike the traditional "transformational" view of programs where the functional relationship between the input state and the output state defines the meaning of a program. Indeed, reactive systems are not dedicated to the transformation of data (like traditional programs), but rather to the control of processes. There are many examples of such concurrent reactive systems: computer networks, asynchronous circuits, operating systems, and various forms of plant-controller systems, such as telephone switches, flight-control systems, manufacturing-plant controllers, etc.

It is difficult to design concurrent reactive systems because they usually have an extremely large number of different behaviors which arise from the combinatorial explosion resulting from all possible interactions between the different concurrent components of the system, and the many possible race conditions that may arise between them. This situation makes the development of concurrent reactive systems an extremely difficult task. Concurrent reactive systems

should therefore be checked for correctness before implementation. This may be either a syntactic check or a dynamic check in which case the system should be executed. Testing is also of very limited help since test coverage is based on only a fraction of the possible behaviors of the system. Since testing is not adequate this situation attracts more importance due to the increasing usage of reactive systems in control safety-critical devices (e.g., flight-control systems) or economically-crucial systems (e.g., telephone switches).

Verification provides the means to ensure the correctness of the design of concurrent reactive systems. Verification means checking that a system description conforms to its expected properties. These properties can range from several forms of consistency to complex correctness requirements specified, for instance, in a logical language. Verification is thus the tool that enables the designer to be confident that the formal description of the system he/she has obtained does indeed satisfy the problem requirements.

Note that "verify" means to (mathematically) prove that a system meets its correctness requirements. We specifically do not mean testing (unless it is exhaustive) or any other method that ensures that the system is "probably" correct. In order to prove that a system conforms to a property, all possible behaviors of the system have to be checked to determine if all of them are compatible with the given property.

State space exploration is one of the most successful strategies for analyzing and verifying finite state concurrent reactive systems. It consists in exploring a global state graph representing the combined behavior of all concurrent components in the system. This is done by recursively exploring all successor states of all states encountered during the exploration, starting from a given initial state, by executing all enabled transitions in each state. The state graph that is explored is called the state space of the system. If the state space is finite, it can be explored completely. If not the methods developed in this thesis may help to restrict the verification to a finite state space which is sufficient for the property checked.

Many different types of properties of a system can be checked by exploring its state space: deadlocks, dead code, violations of user-specified assertions, etc. By the development of model-checking methods for various temporal logics (e.g., [CES86], [LP85], [QS91], [VW86]) the range of properties that state space exploration techniques can verify has been substantially broadened.

State space exploration is a simple strategy due to the easy, and efficient, implementations. Moreover, verification by state space exploration is fully automatic: no intervention of the designer is required. This is a crucial feature for a verification technique to be used in the industry.

Many present verification tools like COSPAN[Kur94], SPIN[Hol91], SDT [SDT95], ObjectGeode[GEO96], CAESAR[FGM<sup>+</sup>92] and POVSDL which we developed use state space exploration. These tools differ by the formal description languages they use for representing systems and properties, and by the conformation criterion according to which these representations are compared. But all of them are based on state space exploration algorithms for performing verification.

Size of the state space enormously increases when using state space exploration verification techniques. Owing to simple combinatorics, the size can be exponential in the size of the description of the system being analyzed. This exponential growth is known as the *state explosion problem*.

The *state explosion problem* is due, among other causes, to the modeling of concurrency by interleaving, in other words, to the exploration of all possible interleavings of concurrent events. For instance, the execution of  $n$  concurrent events is investigated by exploring all  $n!$  interleavings of these events.

## I.2 Partial Order Methods

It is not a priori necessary to explore all interleavings of concurrent events for verification: interleavings corresponding to the same concurrent execution contain similar information. Therefore one can thus hope to be able to verify properties of a concurrent system without exploring all interleavings of its concurrent executions. In this thesis, we present a collection of methods, called *partial order*

*methods* [Gai88, KP92, Pra86, Val90, God95], that make this possible.

The simple idea behind partial order methods is that concurrent executions are really partial orders and that expanding such a partial order into the set of all its interleavings is an inefficient way of analyzing concurrent executions. Instead, concurrent events should be left unordered since the order of their occurrence is not important. The methods we develop use an interleaving representation of partial orders, but attempt to limit the expansion of each partial order computation to just one of its interleavings, instead of all of them.

Precisely, given a property, partial order methods explore only a reduced part of the global state space that is provably sufficient to check the given property. The difference between the reduced and the global state spaces is that all interleavings of concurrent events are not systematically represented in the reduced one.

### **I.3 Overview**

Most realistic system or software specified in a formal language such as SDL defies simple verification techniques simply because of its size. Various techniques to circumvent the complexity of large verification problems have been suggested, some in terms of an informal understanding of the system (informal validation) and others that exploit general and specific aspects of both the system and the property to be verified in order to reduce the effective size of the original formal verification problem. A class of such complexity relief techniques go under the title of *partial order methods*. Based on the idea of trace equivalence suggested by [Maz89], these methods reduce the search space in reachability analysis by discarding a group of transitions at each state with the property that any problematic (or desired) state can be reached by making use of the discarded transitions if and only if it can be reached by avoiding them. In general, safety verification problem can be reduced to that of deadlock checking and methods for verifying a liveness property, though not identical to the safety properties, are not far off so far as adopting the deadlock checking complexity relief techniques

to liveness checking is concerned [Pel94]. This has motivated us to restrict our concerns to deadlock checking only.

In this thesis our approach and results are specific to the structure of Specification and Description Language (SDL) which is a widely used formal description language especially in telecommunications industry. We apply the deadlock checking problem using *persistent sets* [God95] to a class of automata called SSM (SDL-like State Machine) derived from SDL programs. SSM has aspects that are specific to the structure of the SDL language and consequently transition elimination via persistent sets can be made more specific and efficient compared to partial order methods suggested for general concurrent automata models. Indeed this research grew out of concerns for tuning the verifier COSPAN [Kur94] developed in Bell Labs to programs compiled from a subset of SDL into the input language of COSPAN [Kur94], namely S/R [Gla95]. It was then realized that a general verifier for the automaton S/R using partial order methods misses opportunities in complexity reduction unless some of the properties of the SDL language are exploited in a systematic manner. This is where the contribution of this thesis lies : applying the partial order approach to SDL programs by first deriving a necessary and sufficient characterization of transition reduction in terms of persistent sets for SDL programs and then by deriving an algorithm to compute persistent sets based on a decidable version of this characterization.

In [TGH95] and [Tog95] the problem of incorporating partial order methods to generate restricted number of simulations to be used for test case generation have been investigated. We note that our method covers a larger subset of SDL where FIFO discipline of input queues are violated. This condition can be shown to violate the syntactic nature of transition independence as assumed in the cited references. The reduction method called *Condition Locking Simulation* given in [TGH95], is an application of a modified sleep set method given in [God95]. In general the methods for the reductions used by the authors cannot easily be compared to our method using persistent sets since both the purpose (reduced executions vs. deadlock detection) and the models used (SDL exclusive of save and priority inputs vs. inclusive version) in the approaches differ. The basic ideas

presented in this thesis appeared in a technical report [SY97] on top of which the theory has been expanded, several examples, explanatory materials and POVSDL tool implementations have been added.

In the next section there is a brief introduction on SDL which captures the basic functionalities of the language. In chapter III we first describe the formalism called SSM which models the underlying transition system of a subset of SDL by abstracting out the linguistic details of SDL that are useful for a user-friendly specification environment but orthogonal to the analytical concerns used in verification. After the definition of SSM automaton and product SSMs we define the concepts of *independent transitions* and *persistent sets* [God95] in an SSM environment.

In chapter IV we state the basic result on the use of persistent sets for locating deadlocks by restricting the search space to a smaller size. In the second section of chapter IV we apply the persistent set idea to SSM models corresponding to a relatively simplified subset of SDL in which the FIFO discipline of input queues are preserved. We state results that supply alternative and equivalent characterizations of persistent sets in the context of SSM automata, first for individual input transitions then for output and input transitions combined. We then present a decidable version of the persistency characterization for the latter case which is only sufficient but effectively computable. Based on this decidable version we present a recursive algorithm that computes a persistent set at each global state that is minimal relative to the initial transition choice and the computable sufficient condition. In the third section we extend the results of the previous section to a larger SDL set that covers SDL primitives that override the bounds of FIFO reading discipline from the input queues via *save* and *priority input* constructs. In chapter V we explain the tool POVSDL that implements the theory developed in this thesis. The tool consists of a series of modules like scanner and parser for the input language of the tool and a main program implementing the reduction algorithms. In what follows we report our experimental results with real protocol examples one of which is ISDN Layer 2. The results are impressive in the sense that a reduction of order of magnitude has been achieved

for the state spaces generated compared to a conventional reachability analysis and in the final chapter we discuss the ongoing work for future implementations.

# CHAPTER II

## SDL

SDL (Specification and Description Language) is a formal language developed and standardized by CCITT (now ITU-T) for unambiguous specification and description of systems. SDL is mainly known in the telecommunication field, however it can be used for specification of most of reactive and discrete systems as well. The development of SDL began in early 1970's. The latest version is SDL'92 with improvements especially in the area of object orientation on the previous version SDL'88.

In this chapter, an overview of SDL is given. The interested reader is referred to [OFMP<sup>+</sup>94], [FO93] as detailed references and to [SDL92] as the formal standard definition of SDL'92.

SDL is used to describe the structure and behavior of systems. A system can be described using one of two syntactic forms: The textual phrase representation (SDL/PR) and the graphical representation (SDL/GR). Some constructs of SDL have no graphical representation. These are described in textual form which is common to SDL/PR and SDL/GR.

SDL provides different levels of abstraction for the specification of systems. In the most abstract view, there is a system composed of blocks. Blocks are connected to each other and to the environment by means of channels. Each block may be composed of either other blocks or processes. Processes are connected to

each other by means of signal-routes. Communication between the processes is achieved by signals. The signals are transmitted via signal routes and channels. The signals may have parameters to carry information. Communication by means of sending signals is asynchronous.

The channels and signal-routes are defined in SDL as FIFO queues. A signal-route is delay-free whereas a channel may be delay-free or it may have arbitrary delay.

Each process has a certain behavior. The combined behavior of processes in a system constitutes the behavior of the system. Processes in the system execute independently and concurrently. Processes can be viewed as generalized forms of finite state machines with some exceptions. The process body is a flow diagram defining the states and transitions of the process. Each process has an infinite input queue. The signals that are received from other processes are stored in this queue until they are consumed (read). Some basic constructs in SDL for the description of a process are: *start*, *state*, *input*, *priority input*, *output* and *nextstate*.

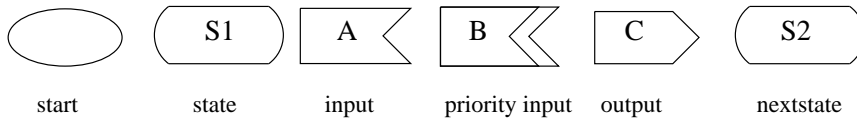


Figure II.1: Basic constructs for the description of a process

The *start* is the initial state of a process. An *input* is an acceptance of a signal by a process at a certain state. It executes if the signal in the input symbol is at the head of the process's input queue. An input symbol connects a state to the actions which the process shall take after consuming the signal mentioned in the input symbol. These actions are among *output*, *task*, *decision*, *procedure call*, *create*, *nextstate*. A *nextstate* symbol implies the end of a transition and beginning of another state.

A process makes transitions between states according to the signals in front of its queue. If a transition is not specified for a particular signal from a state

and it is not in the *save set* of that state then it is implicitly consumed. It is possible to save some signals at some states for future use. Signals which take priority over other signals at a state are placed inside *priority input* symbols. If a state has both ordinary inputs and priority inputs, the first signal in the input queue mentioned in a *priority input* is consumed, even if it is not the first signal in the input queue.

During the transitions, a process may create other process instances by the *create* construct. There may be more than one instance of the same process. Each process instance in the system has a unique identifier of the predefined SDL type PId. The number of instances of a process to be created at system initialization and the maximum number of instances that may exist at the same time are specified by the user.

A process may send signals to other processes or to the environment by an *output* transition. A signal may be sent to a specific process indicated by its PId. If the name of the process is indicated instead of its PId then the signal is sent to an arbitrary instance of the process. If no path is specified, an arbitrary path is chosen among the possible ones.

Each process has its local variables. During the transitions, it may use and change these variables by the use of *task* constructs.

SDL has a timer construct in order to describe time constraints in a specification. Timers are defined within processes. A timer is set with an expiration time by a process. When the time is reached, a signal with the same name as the timer is inserted into the input queue of the process. The timer can be reset by the process if it is no longer needed. If the timer is reset when its associated signal is in the queue, then the signal is removed from the queue.

A process may also call procedures. SDL procedures are similar to the procedures in common programming languages. A procedure may have parameters. The behavior of a procedure is defined in a similar way as the behavior of a process, that is, with states and transitions. A procedure may call another procedure or itself. When a procedure is called, the execution of the caller is suspended until the termination of the procedure. A procedure may share the variables defined

within the calling process.

It is possible to control the flow of execution in a process or procedure by using *decisions*. A decision splits a transition into several possible branches. The decision contains an expression and each branch contains a range for the result of the expression. The ranges must not overlap. The flow of execution continues with the branch whose associated range matches the result of the expression at the time of decision. The *any* operator of SDL can also be used for branching. In this case a branch is chosen arbitrarily which introduces nondeterminism in the behavior.

A process can also be defined by means of SDL services. The whole process behavior is the composition of the partial behaviors of services inside the process. A service body is defined in the same way as a process body. The services of a process share the input queue of the process. The input signal sets of services should be disjoint. Services within the same process cannot make transitions concurrently. According to the signal in the front of the input queue, one service makes a transition and the others wait in a state.

A process can only be terminated by itself. If the process is defined by services then the process terminates after all of its services terminate.

In SDL, there exists a predefined set of data types. In addition to these the user can introduce new data types.

With the new features of SDL'92 in the area of object orientation, it is possible to define types of SDL entities such as system type, block type, process type, service type. The SDL entities in a system can be constructed by using these types. Types can be extended by inheriting another type and adding new properties to it or redefining virtual properties of a type. SDL packages can be defined which consist of several type definitions. These packages can be used in different system specifications.

SDL'92 definition consists of the main recommendation text[SDL92] accompanied by a number of annexes. The main recommendation text is a complete language reference manual. It contains the concrete textual grammar, concrete graphical grammar and abstract grammar for SDL'92. The concrete syntax is

defined in extended Backus-Naur form (BNF) notation and the abstract syntax is defined in Meta-IV language which is a proper part of the Vienna Development Method (VDM)[WH93]. The semantics of SDL is described in informal language in the main text. Annex F to the recommendation contains the precise and detailed static and dynamic semantics of SDL'92 defined in Meta-IV language.

# CHAPTER III

## SDL-LIKE STATE MACHINE (SSM)

In this chapter we define an automaton called SSM that models an SDL process. SSM is, in general, an infinite state automaton since it retains the infinite input queues of SDL. If SDL is restricted to a finite state version by having its input queues bounded - as in the compiler from SDL to S/R [Gla95] - then SSM can readily be made to inherit this property of bounded buffers. Since our concern in this thesis is not the entire verification strategy but the specific form of persistent sets for communicating SSM automata we shall assume that neither SDL nor its simplified model SSM are restricted to finite automata. Although we explain the mapping of SDL into SSM in terms of SDL without the data part, the results of the thesis apply to the extended case. In next chapter we explain roughly how our results can be applied to an SDL system incorporating data. But in any case, we avoid the use of revealed/viewed variables which is not recommended even by the authors of SDL for new system specifications, in an informal way through [OFMP<sup>+</sup>94] although it is still a part of the language formally. For simplicity of an initial exposure the treatment in this chapter ignores SDL constructs such as *save* and *priority inputs* that violate the FIFO principle. This is generalized in the next chapter. In section III.2 of this chapter we will give the definition of independency of transitions and definition of traces. These will help us to find equivalent transition sequences and in the next chapter this background will help

us to model partial order methods.

### III.1 SSM

An SSM automaton  $P$  is a quadruple  $P = (X, M, T, x_o)$  where

1.  $X = C \times Q$  is the state set where the finite set  $C$  is called the control point component and  $Q = M_+^*$  is an unbounded queue component of received messages the members of which belong to a finite set  $M_+$ . We use the notation  $x_c$  and  $x_q$  to refer to the components of the state  $x$ .
2.  $M = (M_+ \cup M_-)$  is a finite set of messages that  $P$  can receive and send respectively.
3.  $T \subseteq X \times M \times X$  is a set of transitions. We use the notation  $x \xrightarrow{m} x'$  to mean  $t = (x, m, x') \in T$  where, in addition we let  $pre(t) = x$ ,  $post(t) = x'$  and  $sig(t) = m$  to denote the source and target states and the message content of a transition  $t$ . By a slight abuse of notation we shall also write  $x \xrightarrow{t} x'$  for a transition  $t$  instead of  $x \xrightarrow{m} x'$  whenever the contents  $m = sig(t)$  of the message is not of any concern.
4.  $x_0 \in X$  is the initial state of  $P$ .

We next explain the semantic constraints of the transition structure of an SSM arising from the corresponding constraints of SDL and then define product of SSM automata where SDL communication takes place via message exchanges. For this we first describe the transition structure  $T$  of an SSM automaton.

The transition set  $T$  is composed of three types of transitions denoted by sets of *receiving transitions*  $T^{rc}$ , *reading transitions*  $T^{rd}$  and *output transitions*  $T^{out}$ . Therefore we can express the set  $T$  in terms of the set union  $T = T^{rc} \cup T^{rd} \cup T^{out}$ . We explain the generic constraints associated with each kind of transition below.

- (1)  $t = (x, m, x') \in T^{rc}$  iff  $x_c = x'_c$ ,  $x'_q = x_q.m$  and  $m \in M_+$ ,
- (2)  $t = (x, m, x') \in T^{rd}$  implies that  $x_q = m.x'_q$  and  $m \in M_+$ ,
- (3)  $t = (x, m, x') \in T^{out}$  implies that  $x'_q = x_q$  and  $m \in M_-$  and  $t'' = (y, m, y') \in T^{out}$  for any  $y$  subject to  $y_c = x_c$ .

In addition to the transition constraints above the following two additional SDL - induced constraints related to deterministic input and output behaviour prevail:

- (a) If  $(x, m, y), (x', m, y') \in T^{rd}$  and  $x_c = x'_c$  then  $y'_c = y_c$  (strict input determinism of SDL).
- (b) If  $(x, m, y), (x', m', y') \in T^{out}$  and  $x_c = x'_c$  then  $m' = m$  and  $y'_c = y_c$  (strict output determinism of SDL).

It is clear from (3) and (b) above that at a control point of output transition there cannot be any other transition. As a suggestive notation we write  $-m$  or  $+m$  to mean  $m \in M_-$  or  $m \in M_+$  respectively. Following the usual convention we extend the notation for one-step transitions to its transitive closure and write  $x \xrightarrow{s} x'$  where  $s \in M^*$  denotes a multi-step transition composed of any combination of transitions.

Finally we explain the composition of SSM automata or processes <sup>1</sup>. The only manner in which individual SSM processes synchronize is when an output transition in  $T_i^{out}$  of a process  $P_i$  synchronizes with a receiving transition in  $T_j^{rc}$  of another process  $P_j$ . In order to specify in SSM notation those outputs and inputs that synchronize we let  $receiver(sig(t))$  denote the (integer) index of the process that receives the signal  $sig(t)$  of an output transition  $t$  at the instance it is output by a process. Although it is possible to model the channel routing non-determinacy of SDL communication in an SSM environment, we shall nevertheless assume that the destination address of every output operation is deterministic.

The product automaton composed of individual (single input queue) SSM automata is not an SSM automaton since it has multiple queues. Formally if the individual SSM automata are  $P_i = (X_i, M_i, T_i, x_{0i})$  then  $P := P_1 \times \dots \times P_n := (G, M, T, g_0)$  where  $G = (X_1 \times \dots \times X_n)$ ,  $M := \cup_{i=1,n} M_i$ ,  $T \subseteq G \times M \times G$  and  $x_0 := (x_{01}, \dots, x_{0n}) \in G$  where  $T$  is defined to be the smallest set that satisfies the conditions below.

- (i) If  $(g[i], +m_i, g'[i]) \in T_i^{rd}$  then  $(g, +m_i, g') \in T$ , where  $g'[j] = g[j]$  whenever  $j \neq i$ , that is, every local input reading transition is also a global transition where the remaining processes do not change states,
- (ii) If  $(g[i], -m_i, g'[i]) \in T_i^{out}$  and  $k = receiver(-m_i)$  then  $(g, -m_i, g') \in$

---

<sup>1</sup> We use the terminology *automaton* and *process* in an interchangeable manner.

$T$  where  $g'_c[k] = g_c[k]$ ,  $g'_q[k] = g_q[k]$ . -  $m_i$  and  $g'[j] = g[j]$  whenever  $j \neq i, k$ .

The definition of global transition above implies that every global transition is based on an input or output local transition. We shall call this local transition the *local version* of the global transition. Conversely corresponding to every input or output local transition there is a global transition that is enabled at any global state which is consistent - i.e., in conformity with the definition above - with the local state that defines the local transition. We shall often refer to a global transition defined at a global state  $g$  as the *global version* of the local transition  $t$ . By an abuse of notation we may use  $t$  to denote both the local and global versions of a transition.

We say that a transition  $t$  - more precisely the global version of a local transition  $t$  - is *enabled* at  $g$  if there exists  $g'$  where  $g \xrightarrow{t} g'$ . We write  $g^k \rightarrow g^l$  if there exists a transition  $t$  such that  $g^k \xrightarrow{t} g^l$ . Let  $\rightarrow^*$  denote the reflexive and transitive closure of the relation  $\rightarrow$ . A state  $g'$  is *reachable from* a state  $g$  iff  $g \rightarrow^* g'$ . The global reachability set of composed SSMs is the set of all states reachable from the initial state  $g_0$ .

It should be clear to the reader by now that, an output statement in SDL will be represented by an SSM transition in  $T^{out}$  together with its synchronizing counterpart SSM transition in  $T^{rc}$ , and an input statement in SDL will correspond to an SSM transition in  $T^{rd}$ . Later in this thesis, we enlarge the SDL subset handled, and therefore add new features to SSM model. The new SDL constructs added will correspond, in a similar direct way, to the new features added to SSM.

## III.2 Independency and Traces

**Definition 1** *Two local transitions  $t_1$  and  $t_2$  are said to be independent at a global state  $g$  iff their global versions are both enabled at  $g$  and*

1.  $g \xrightarrow{t_1} g'$ , then  $t_2$  is enabled in  $g'$  and vice-versa when  $t_1$  and  $t_2$  are interchanged (independent transitions cannot disable each other)

2. *there exists a unique state  $g'$  such that  $g \xrightarrow{t_1 \cdot t_2} g'$  and  $g \xrightarrow{t_2 \cdot t_1} g'$  (commutativity of independent transitions)*

**Remark 1** *It can be easily seen that for all global states  $g$  :*

1. *Enabled local transitions  $t = (g[i], +m, g'[i]) \in T_i^{rd}$  and  $t' = (g[j], +m', g''[j]) \in T_j^{rd}$  for  $i \neq j$  are independent at  $g$  .*
2. *Enabled local transitions  $t = (g[i], +m, g'[i]) \in T_i^{rd}$  and  $t' = (g[j], -m', g''[j]) \in T_j^{out}$  are independent at  $g$ .*
3. *Enabled local transitions  $t = (g[i], -m, g'[i]) \in T_i^{out}$  and  $t' = (g[j], -m', g''[j]) \in T_j^{out}$  for  $i \neq j$  are independent at  $g$  if and only if  $receiver(-m) \neq receiver(-m')$  , i.e. two output transitions are independent iff they send messages to different processes.*

Following the work of Mazurkiewicz [Maz89], one can use the notion of independent transitions to define an equivalence relation on sequences of transitions: *two sequences of transitions are equivalent if they can be obtained from each other by successively permuting adjacent independent transitions.* Thus, given a valid dependency relation, sequences of transitions can be grouped into equivalence classes which Mazurkiewicz calls *traces* which are formally defined as follows [Maz89].

**Definition 2** *A concurrent alphabet is a pair  $\Lambda = (T, D)$  where  $T$  is a finite set of transitions, called the alphabet of  $\Lambda$ , and where  $D$  is a binary, reflexive, and symmetric relation on  $T$  called the dependency in  $\Lambda$ .*

The relation  $I_\Lambda = T^2 \setminus D$  represents the independency in  $\Lambda$ .

**Definition 3** *Let  $\Lambda = (T, D)$  be a concurrent alphabet, let  $T^*$  represent the set of all finite sequences (words) of symbols in  $T$ , let  $\cdot$  stand for the concatenation operation, and let  $\epsilon$  denote the empty word. We define the relation  $\equiv_\Lambda$  as the least congruence in the monoid  $[T^*; \cdot; \epsilon]$  such that*

$$(t_1, t_2) \in I_\Lambda \Rightarrow t_1 t_2 \equiv_\Lambda t_2 t_1$$

The relation  $\equiv_{\Lambda}$  is referred to as the *trace equivalence over  $\Lambda$* .  $[T^*; \cdot; \epsilon]$  is a monoid in which the concatenation operation  $\cdot$  may be commutative for some pairs of different elements. It is sometimes called a free partially commutative monoid over  $T$ .

**Definition 4** *Equivalence classes of  $\equiv_{\Lambda}$  are called traces over  $\Lambda$ .*

The trace containing a sequence of transitions  $\omega$  will be denoted  $[\omega]_{(T,D)}$  or  $[\omega]$  for short when there is no ambiguity. A trace is fully characterized by one of its sequences  $\omega$  and a concurrent alphabet  $\Lambda = (T, D)$ : by successively permuting adjacent independent transitions in  $\omega$ , one can obtain all the other sequences in  $[\omega]$ . The proof of the following theorem is simple and can be found in [God95].

**Theorem 1** *Let  $g$  be a state in  $A_G$ . If  $g \xrightarrow{\omega_1} g_1$  and  $g \xrightarrow{\omega_2} g_2$  in  $A_G$ , and if  $[\omega_1] = [\omega_2]$ , then  $g_1 = g_2$ .*

It is important to notice that certain transitions (i.e. which are independent) constitute equivalent transition sequences which are *traces*. The above definitions and theorems about independency of transitions and traces will be the building blocks of our partial order methodology developed in the next chapter.

# CHAPTER IV

## REDUCED SSM BY PERSISTENT SETS

In this chapter we define the concepts that eventually lead to a complexity relief result used for restricting the state space. We start with the definition of persistent sets by the help of which we will show that at a global state of a system it will suffice to find deadlocks if we move only with the transitions taken from persistent sets. We will apply the persistent set idea to a basic case which covers a subset of SDL, then we will apply the idea to an extended case which includes priority and save signal concepts of SDL. A sufficient algorithm will be given to find persistent sets. We will finally explain how we can handle data in SDL by the methodologies developed.

### IV.1 Persistent Sets

The usual way to check the existence of deadlocks is to generate the set of reachable states. The complexity problem of reachability analysis is caused by the size of the search space. The idea in partial order methods is to restrict the size of the search space by the elimination of redundant and excessive interleaving possibilities of component process transitions. This is captured in the definition of *persistent sets* [God95] defined below.

**Definition 5 (Persistent sets)** *A set  $T$  of transitions enabled at a global state  $g$  is said to be persistent at  $g$  iff for all non-empty sequence of transitions of the*

```

Initialize: Stack is empty; Hash table H is empty;
           push(s0) onto Stack;
Loop : while Stack is not empty do{
           pop (s) from Stack;
           if s is NOT already in H then {
               enter s in H;
               T= Persistent_Set(s);
               for all t in T do {
                   s'=successor(s) after t; /*t is executed*/
                   push(s') onto Stack;
               }
           }
       }

```

Figure IV.1: State space generation with Persistent set

form

$$g = g_1 \xrightarrow{t^1} g_2 \dots g_n \xrightarrow{t^n} g_{n+1}$$

with  $t^i \notin T$ ,  $t^n$  is independent with all the transitions in  $T$  at state  $g_n$ .

Intuitively, a subset  $T$  of the set of transitions enabled in a state  $g$  of product automaton  $A_G$  is called *persistent in  $g$*  if all transitions not in  $T$  that are enabled in  $g$ , or in a state reachable from  $g$  through transitions not in  $T$ , are independent with all transitions in  $T$ . In other words, whatever one does from  $g$  while remaining outside of  $T$ , does not interact with or affect  $T$ . Note that the set of all enabled transitions in a state  $g$  is trivially persistent since nothing is reachable from  $g$  by transitions that are not in this set.

Let  $T_g$  be persistent sets defined at every reachable state  $g$  of the product automaton  $A_G$  and let the automaton  $A_R$  be the reduced automaton obtained from  $A_G$  by deleting all the transitions  $t \notin T_g$  at each state  $g$ .  $A_R$  can be obtained by the algorithm illustrated in Figure IV.1 with a depth first search. The algorithm recursively explores all successor states of all states encountered during the search, starting from the initial state, by executing all elements of persistent set in each state. The main data structures used are a 'Stack' to store the states whose successors still have to be explored, and a hash table 'H' to store all the states that have already been visited during the search. The set of all transitions that are persistent in a state 's' is denoted by 'Persistent\_Set(s)'.

The state reached from a state 's' after the execution of a transition 't' is denoted 'successor(s) after t'.

**Lemma 1** *Let  $g$  be a state in  $A_R$ , and let  $d$  be a deadlock reachable from  $g$  in  $A_G$  by a nonempty sequence  $\omega$  of transitions. For all  $\omega_i \in [\omega]_g$ , let  $t_i$  denote the first transition of  $\omega_i$ . Let  $\text{Persistent\_Set}(g)$  be a nonempty persistent set in  $g$ . Then, at least one of the transitions  $t_i$  is in  $\text{Persistent\_Set}(g)$ .*

Proof of Lemma:

Let the sequence  $\omega$  of transitions be  $t_1 t_2 \dots t_n$ , and let  $g = g_1 \xrightarrow{t_1} g_2 \xrightarrow{t_2} g_3 \dots \xrightarrow{t_{n-1}} g_n \xrightarrow{t_n} d$  be the sequence of states it goes through in  $A_G$ . Assume first that none of the transitions in  $\omega$  are in  $\text{Persistent\_Set}(g)$ . Then, by persistent set definition, for all transitions  $t_j$ ,  $1 \leq j \leq n$ ,  $t_j$  is independent in  $g_j$  with all transitions in  $\text{Persistent\_Set}(g)$ . Thus, by definition of independent transitions, all transitions in  $\text{Persistent\_Set}(g)$  remain enabled in all states  $g_j$ ,  $1 \leq j \leq n$ , and in  $d$ , which hence cannot be a deadlock. Thus, some transition of the sequence  $\omega$  from  $g$  to  $d$  must be in  $\text{Persistent\_Set}(g)$ .

Let thus  $t_k$  be the first transition in  $\omega$  that is in  $\text{Persistent\_Set}(g)$  and let  $\omega'$  be the sequence  $t_k t_1 \dots t_{k-1} t_{k+1} \dots t_n$ , i.e., the sequence  $\omega$  where the transition  $t_k$  is moved to the first position. By definition of persistent sets, we have that for all  $1 \leq j < k$ ,  $t_j$  is independent with  $t_k$  in  $g_j$ . Consequently, by definition of a trace,  $\omega' \in [\omega]_g$ , and the lemma is proved. *QED*

**Theorem 2** *Let  $g$  be a state in  $A_R$ , and let  $d$  be a deadlock reachable from  $g$  in  $A_G$  by a sequence  $\omega$  of transitions. Then,  $d$  is also reachable from  $g$  in  $A_R$ .*

Proof of Theorem :

The proof proceeds by induction on the length of  $\omega$ . For  $|\omega| = 0$ , the result is immediate. Now, assume the theorem holds for paths (sequences of transitions) of length  $n \geq 0$  and let us prove that it holds for paths  $\omega$  of length  $n + 1$ .

Assume a deadlock  $d$  can be reached from a state  $g$  by a path  $\omega$  of length  $n + 1$  in  $A_G$ . For all  $\omega_i \in [\omega]_g$ , let  $t_i$  denote the first transition of  $\omega_i$ . Let  $\text{Persistent\_Set}(g)$  be the nonempty persistent set that is selected in  $g$  by the algorithm

of Figure(IV.1) i.e., the set of transitions that are explored from  $g$  in  $A_R$ . From Lemma (1), we know that at least one of the transitions  $t_i$  is in  $\text{Persistent\_Set}(g)$ . Since  $t_i$  is in  $\text{Persistent\_Set}(g)$ , it is explored from state  $g$  and a state from which a path of length  $n$  leads to the deadlock  $d$  is reached in  $A_R$ . This together with the inductive hypothesis proves the theorem. *QED*

From the theorem above it is then immediate to conclude that a persistent set selective search started in the initial state of  $A_G$  will explore all deadlocks in  $A_R$ .

In verifying SDL programs two distinct but related concepts are involved. The first is a straightforward SDL deadlock where each process halts at an input state and either the queues are empty or contents are in the save lists. The second is what is known as *unspecified reception* where an unexpected message lies at the head of the queue. Commercial verifiers detect such unspecified reception situations as well as deadlocks. In SSM terms the latter can be modeled as a straight deadlock or if it is to be ignored, self looping read statements can be inserted to consume unspecified messages at the head of the queues in conformity with the semantics of SDL.

## IV.2 The Basic Case

### IV.2.1 Reduction theorems

A global state where there exists at least one enabled input read transition has the attractive property that any one of these enabled input read transitions can be used as a (singleton) persistent set at that state. The related theorem follows.

**Theorem 3 (Read-first principle)** *Every input read transition  $t \in T$  enabled at a global state  $g$  is a singleton persistent set at  $g$ .*

Proof of Theorem:

Suppose that  $T$  is not persistent in  $g$ . Thus, by definition of persistent sets, there exists in  $A_G$  a sequence  $g = g^1 \xrightarrow{t^1} g^2 \dots g^n \xrightarrow{t^n} g^{n+1}$  of transitions  $t^1, \dots, t^n \notin T$ ,

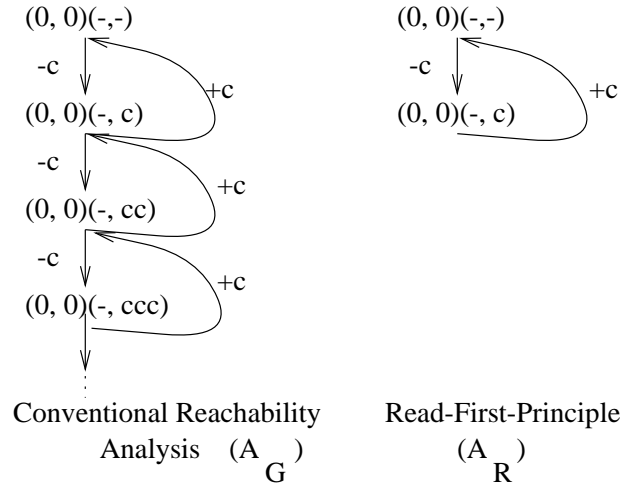


Figure IV.2: Reachable states generated with two methods

such that  $t^n$  is dependent in  $g^n$  with transition  $t \in T$ . Consider the shortest such a sequence. For this sequence, not only  $t^n$  is dependent in  $g^n$  with transition  $t \in T$ , but also, for all  $1 \leq i < n$ ,  $t^i$  is independent in  $g^i$  with transition in  $T$ . Let us show that such a sequence can not exist. From the remark following the definition of independency of transitions it is clear that there cannot be any transition that is dependent with  $t$  in  $g$ . Hence the theorem is proved. *QED*

Figure IV.2 shows  $A_G$  and  $A_R$  respectively for a system of two processes one continuously sending a signal  $c$  to the other and the other is always consuming the signal  $c$ . It is clear that  $A_G$  is infinite whereas  $A_R$  has only two states. The above theorem tells us that at a global SSM state if we have a process that can read a signal from its input queue, then we can expand the reachability tree with this transition and ignore all the other enabled transitions. We now generalize computation of persistent sets to incorporate cases other than singleton input transition sets. Such a computation becomes essential if no component of the global state is at an input reading state with a non-empty queue. We start by stating a theorem on equivalent characterization of persistent sets for an SSM model  $P$ .

**Theorem 4** *Let  $g$  be a global state of the system  $P$ , let  $O$  and  $I$  be the index sets of those processes that are at their output states and their input states with non-empty queues at the global state  $g$ . Let  $O' \subseteq O$  and  $I' \subseteq I$ . Then the set of transitions  $T := \{t_j\}_{j \in I' \cup O'}$  is a persistent set at the global state  $g$  if and only if condition C1 below holds :*

Condition C1 :

For every sequence of transitions starting from  $g$ , that is,

$$g = g^1 \xrightarrow{t^1} g^2 \dots g^m \xrightarrow{t^m} g^{m+1} \quad (\text{IV.1})$$

subject to the constraint that  $t^n \notin T$  for all  $n = 1, \dots, m$ , if  $k := \text{receiver}(\text{sig}(t_j))$  for some  $j \in O'$  then none of the transitions  $t^n$  is based on a receiving input transition  $T_k^{rc}$ , that is, no new message is added to the input queue of the process  $P_k$ .

Proof of Theorem :

Suppose that condition C1 above is violated. Then there exists a sequence of executions given by (IV.1) subject to the constraint that the transitions  $t^n$  are not in  $T$  such that  $g_q^{m+1}[k] = g_q^m[k].\text{sig}(t^m)$  where  $j \in O'$  and  $k = \text{receiver}(\text{sig}(t_j))$ . But then persistency of  $T$  is violated since  $t_j \in T$  and  $t^m$  are not independent at the global state  $g^m$  since  $g^m \xrightarrow{t^m, t_j} g'$ ,  $g^m \xrightarrow{t_j, t^m} g''$  where  $g'_q[k] = g_q^m[k].\text{sig}(t^m).\text{sig}(t_j)$  whereas  $g''_q[k] = g_q^m[k].\text{sig}(t_j).\text{sig}(t^m)$  proving that  $g' \neq g''$ .

Conversely suppose that  $T$  is not a persistent set. Then by definition there is an execution sequence given by (IV.1) with  $t^n \notin T$  for  $n = 1, \dots, m$ , such that  $t_j$  and  $t^m$  are dependent (not independent) at  $g^m$ . By the remark following the definition of independency of transitions and the fact that  $t_j$  is an output transition the only manner in which dependency can occur is when either both  $t^m$  and  $t_j$  are output transitions to a common process, namely  $P_k$  where  $k := \text{receiver}(\text{sig}(t_j))$  ; or  $t^m$  is an input transition of process  $P_k$  with an empty queue at  $g^m$ . The second possibility is void since the input transition  $t^m$  cannot occur with an empty queue at  $g^m$ . On the other hand the first possibility clearly violates C1 since  $t^m$  adds a message to the queue of the process  $P_k$ . *QED*

The theorem above has limited utility since checking whether C1 holds or not is, in general, an undecidable problem due to the infinite queues occurring in

SSM. Even if we restrict the queue sizes of SDL processes - hence their models in SSM - to finite lengths, the complexity of checking  $C1$  may well be unacceptable. Therefore we need a simpler condition to replace  $C1$  at the expense of losing its necessity.

#### IV.2.2 Algorithm

Assume that the definitions of global state  $g$  and the index set  $O' \subseteq O$  and  $I' \subseteq I$  are as in the statement of the theorem above. We define a directed graph  $Gr$  derived from the system  $P$  such that nodes of  $Gr$  are processes of  $P$  and there is an arc from process  $P_k$  to  $P_l$  if there is an output transition of process  $P_k$  that sends a message to the input queue of  $P_l$ . Let  $K(O')$  be the index set of those processes that receive the output messages of processes  $P_j$  with  $j \in O'$  at the global state  $g$ . The following theorem characterizes a sufficient condition for condition  $C1$  to hold.

**Theorem 5** *Suppose that in the graph  $Gr$  above there does not exist a path that starts at a node in  $P_m$  with  $m \in (I \setminus I') \cup (O \setminus O')$  and ends at any process  $P_k$  where  $k \in K(O')$  and none of the nodes other than possibly the first and the last ones of the path have process indices in  $I \cup O$ . Under these conditions the condition  $C1$  of Theorem 4 is valid.*

Proof of Theorem :

We prove the desired result by contradiction. Hence assume that  $C1$  does not hold. Then there exists an execution sequence starting from the global state  $g$  and with the properties given in  $C1$ . Since  $C1$  is violated, without loss of generality, we take the last transition  $t^m$  as an output transition sending a message into the input queue of a process with an index, say  $k$ , in  $K(O')$ . Since by assumption each transition  $t^j$  is not in  $T$  there are two possibilities :  $t^m$  is an output transition of a process with an index, say  $p$ , in  $(I \setminus I') \cup (O \setminus O')$  in which case we are done since the condition of the theorem is violated by existence of the path from  $P_p$  to  $P_k$  facilitated by the output transition  $t^m$  of  $P_p$ ; or the index  $p$  of the process that generates the transition  $t^m$  is not in  $(I \setminus I') \cup (O \setminus O')$  and

thus it is also not in  $I \cup O$  by the assumption that  $t^m \notin T$ . Therefore it must be an output transition of a process  $P_p$  which was at a local input state with its input queue empty at the starting state  $g$ . Let  $n$  be the largest index value for which the transition  $t^n$  is an output transition of a process  $P_{p'}$  sending an input message to the input queue of the process  $P_p$ . That such a transition  $t^n$  must occur follows from the fact that unless input queue of  $P_p$  is filled with a signal it cannot progress and generate the transition  $t^m$ . We apply the same argument used for  $P_p$  to the process  $P_{p'}$  etc., which eventually generates a path starting from a process with index in  $(I \setminus I') \cup (O \setminus O')$  and ending at a process with an index in  $K(O')$ . Clearly this violates the condition given by the theorem which proves the desired result. *QED*

The algorithm for generating persistent sets relies on the proof of Theorem 5. We start from any single member of the index set  $O$  for the set  $O' \cup I'$  at a global state  $g$  and check the condition given by Theorem 5<sup>1</sup>. The convergence of the algorithm and the minimality property of the persistent set relative to the condition of Theorem 5 is obvious and will not be elaborated any further.

### Algorithm 1

**Step 1.** Choose  $I' = \emptyset$  and  $O'$  any singleton subset of  $O$ .

**Step 2.** Check conditions of Theorem 5 with current values of  $I'$  and  $O'$ . If satisfied stop, index set  $I' \cup O'$  corresponds to a persistent set. Else go to step 3.

**Step 3.** For each distinct path in  $Gr$  violating Theorem 5 add the index of its source node to  $I'$  or  $O'$  whichever the case may be and go to step 2.

## IV.3 Extensions

The results of the previous section are now generalized to a larger subset of SDL.

This subset incorporates the *save* primitive and more importantly it covers the

---

<sup>1</sup> Note that if we choose the initial value of  $O'$  empty and  $I'$  a singleton set then the algorithm will immediately return  $\{t_i \mid i \in I'\}$  as a persistent set by virtue of Theorem 3. Hence there is no point in starting from a process index at an input state.

*priority input* construct of SDL.

There are two modifications that must take place after the extension of SDL with these primitives. The first one involves the necessary changes that must be made in the SSM model of an SDL process. The second one is more complex and it involves the new characterization of persistent sets under priority inputs.

Both the save primitive and priority inputs induce a new discipline in reading from the input queues of a process violating the FIFO (first-in-first-out) principle. For this we modify the local state  $g_c[i]$  at a global state  $g$  by associating with it two sets of input messages that are subsets of  $M_{i+}$  called the save subset (denoted  $save(g_c[i])$ ) and the priority subset (denoted  $prior(g_c[i])$ ). In terms of this artifact the rules for global transitions remain intact. The only modification for local transitions is for the read transitions in  $T^{rd}$ , which has to be modified in the way expressed below.

(1)  $t = (x, m, x') \in T^{rd}$  and  $m \notin prior(x_c)$  implies that

$$\begin{aligned}
 x_q &= Y.m.Y', \\
 x'_q &= Y.Y', \\
 [Y] &\subseteq save(x_c), \\
 m &\notin save(x_c), \\
 m &\in M_+, \\
 prior(x_c) \cap [x_q] &= \emptyset
 \end{aligned} \tag{IV.2}$$

(2)  $t = (x, m, x') \in T^{rd}$  and  $m \in prior(x_c)$  implies that

$$\begin{aligned}
 x_q &= Y.m.Y', \\
 x'_q &= Y.Y', \\
 [Y] \cap prior(x_c) &= \emptyset, \\
 m &\notin save(x_c), \\
 m &\in M_+,
 \end{aligned} \tag{IV.3}$$

where  $Y$  and  $Y'$  denote queues and  $[Y]$  denotes the message contents of a queue  $Y$ . Independency of transitions and persistent set characterization are not affected by the inclusion of save construct. The following concept turns out to be instrumental in persistent set characterization when priority inputs are present.

**Definition 6** *Let  $g$  be a global state, then an enabled transition  $t$  at  $g$  is called a priority pre-empting transition relative to a process  $P_k$  if  $t$  is an output transition*

of some component process,  $k = \text{receiver}(\text{sig}(t))$ ,  $\text{sig}(t) \in \text{prior}(g_c[k])$  and the input queue  $g_q[k]$  is nonempty and has no priority signals (messages) that are members of the set  $\text{prior}(g_c[k])$ .

Observe that whenever a priority pre-empting transition occurs at a global state  $g$  the receiving process  $P_k$  cannot read the non-priority signal contents of its queue after the event of receiving the priority signal whereas it could read them before the event. This implies that the transitions  $t$  and  $t_k$  of reading a non-priority signal by  $P_k$  are dependent since  $t_k.t$  can occur but the converse sequence  $t.t_k$  cannot occur. This simple observation is the basis of the additional complications occurring for the extended case as expressed by the counter-parts of the theorems 3 and 4 of the previous section.

**Remark 2** *The statements made by Remark 1 remain true except for the case corresponding to the independence of an output and an input event given by IV.3 which should now read as : the local transitions  $t = (g[i], +m, g'[i]) \in T_i^{\text{rd}}$  and  $t' = (g[j], -m', g'[j]) \in T_j^{\text{out}}$  are independent at  $g$  provided that  $t'$  is not a pre-empting transition at  $g$  relative to  $P_i$ .*

### IV.3.1 Reduction theorems

The first change comes with the input singleton persistent sets. We omit the proof since it is similar to the proof of Theorem 3.

**Theorem 6** *Let  $g$  be a global set and  $t$  be an input read transition based on a local transition  $(g[i], \text{sig}(t), g'[i]) \in T_i^{\text{rd}}$  enabled at  $g$  with the property that either  $\text{prior}(g_c[i]) = \emptyset$  or  $\text{sig}(t) \in \text{prior}(g_c[i])$ . Then the singleton set  $\{t\}$  is a persistent set.*

Next we present the counter-part of Theorem 4 for the extended case. Let  $g$  be a global state and let  $O, I$  and  $J$  denote the process index sets corresponding to the processes at their corresponding local output states ; at their local input states with non-empty priority sets and non-empty input queues with no messages

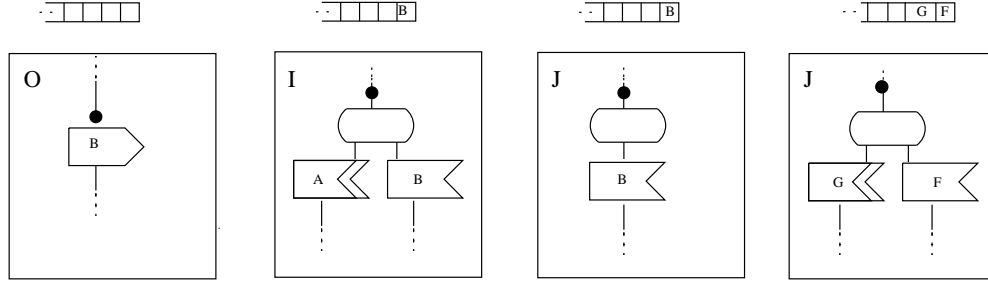


Figure IV.3:  $O, I$  and  $J$  type processes

in priority sets; at their local input states but violating the previous condition respectively. Figure IV.3 makes clear  $O, I$  and  $J$  type processes.

**Theorem 7** *Let  $O' \subseteq O$ ,  $I' \subseteq I$  and  $J' \subseteq J$ . Define the set of transitions  $T := \{t_j\}_{j \in O' \cup I' \cup J'}$  where if  $j \in O'$ ,  $t_j$  denotes the unique output transition and if  $j \in I' \cup J'$ ,  $t_j$  denotes the unique input transition dictated by the head of the input queue of the process  $P_j$ . The set  $T$  is a persistent set if and only if the condition  $C2$  below holds :*

Condition  $C2$ :

Given any execution sequence

$$g = g^1 \xrightarrow{t^1} g^2 \dots g^m \xrightarrow{t^m} g^{m+1} \quad (\text{IV.4})$$

subject to the constraint that  $t^p \notin T$  for all  $p = 1, \dots, m$  :

- (1) A transition  $t_j$  with  $j \in O'$  is not a priority pre-empting transition at some  $g^p$  relative to a process  $P_i$  with  $i \notin O' \cup I' \cup J'$ ,
- (2) A transition  $t^p$  is not a priority pre-empting transition at  $g^p$  relative to a process  $P_i$  with  $i \in I'$ ,
- (3) If

$$K(O') := \{\text{receiver}(\text{sig}(t_j)) \mid j \in O'\} \quad (\text{IV.5})$$

then no new message is added to the input queue of  $P_k$  via any of the transitions  $t^p$  where  $k \in K(O')$ .

Proof of Theorem :

Suppose that condition  $C2$  is violated. Then there exists an execution sequence given by (IV.4) and three possibilities exist :

- (i) Item (1) of  $C2$  is violated. Then the associated output transition  $t_j$  for some

$j \in O'$  is priority pre-empting at some  $g^p$  relative to a process  $P_i$  where  $i \notin O' \cup I' \cup J'$ . But by the remark following the definition of a priority pre-empting transition this implies that  $t_j$  is not independent with the input transition  $t'$  of  $P_i$  at  $g^p$ . This clearly violates the definition of persistency of  $T$  since  $t_j \in T$  and  $t' \notin T$ .

(ii) Item (2) in  $C2$  is violated. Then  $t^p$  is a priority pre-empting transition at  $g^p$  relative to some  $P_i$  with  $i \in I'$ . Similar to the previous case this implies that the input transition  $t' \in T$  of  $P_i$  and  $t^p \notin T$  are dependent and therefore  $T$  is not a persistent set.

(iii) Item (3) of  $C2$  is violated. Then the proof follows the identical argument to the first part of proof of Theorem 4 and will not be repeated.

Conversely suppose that  $T$  is not a persistent set. Then there exists an execution sequence given by (IV.4) where  $t^m$  is dependent on some  $t_j \in T$  at  $g^m$ . We investigate the nature of this dependency case by case.

Case 1. Both  $t^m$  and  $t_j$  are output transitions. Then dependency can only occur when  $k := receiver(sig(t^m)) = receiver(sig(t_j))$  which implies that item (3) of  $C2$  is violated since  $g_q^{m+1}[k] = g_q^m[k].sig(t^m)$ .

Case 2.  $t^m$  is an input transition and  $t_j$  is an output transition. Then by Remark 2 dependency can occur iff  $t_j$  is a priority pre-empting transition at  $g^m$  relative to the process executing the input transition  $t^m$ . This clearly violates item (1) of  $C2$ .

Case 3.  $t^m$  is an output transition whereas  $t_j$  is an input transition. Again by Remark 2 dependency can occur iff  $t^m$  is a priority pre-empting transition relative to  $P_j$  which violates item (2) of  $C2$ .

Case 4. Both  $t^m$  and  $t_j$  are input transitions. This is not possible since two input transitions of two distinct processes are always independent and since  $t^m \notin T$  and  $t_j \in T$  these input transitions indeed belong to distinct processes.

This completes the proof of the theorem. *QED*

### IV.3.2 Algorithm

In order to give a decidable version of  $C2$  we again make use the graph  $Gr$  given in the previous section.

**Theorem 8** *Suppose that there does not exist a path in  $Gr$  that starts from any  $P_i$  with  $i \in (O \setminus O') \cup (I \setminus I') \cup (J \setminus J')$  with all the intermediate nodes with indices not in  $O \cup I \cup J$  and the final node is  $P_j$  where either :*

(i)  $j \in K(O')$  where  $K(O')$  is given by the expression (IV.5) or ;

(ii)  $j \in I'$  and the process just before  $P_j$  on the path has an output transition  $t$  with  $sig(t) \in prior(g_c[j])$  or;

(iii)  $j \notin O' \cup I' \cup J'$ ,  $P_j$  has an input state with a non-empty priority set including  $sig(t_k)$  for some  $k \in O'$ ,

then condition (C2) of Theorem 7 is satisfied.

Proof of Theorem:

We prove the assertion by contradiction. Suppose that  $C2$  is violated, then there exists an execution sequence given by (IV.4) which violates at least one of the 3 items of  $C2$ .

If item (1) is violated then some  $t_i$  is priority pre-empting at some state  $g^p$  relative to a process  $P_j$  with  $j \notin O' \cup I' \cup J'$ . This clearly violates hypothesis (iii) of the Theorem. Suppose now that the sequence violates item (2). Then some output transition  $t^p$  in this sequence is priority pre-empting at  $g^p$  relative to some  $P_i$  with  $i \in I'$ . If  $P_k$  is the process generating the output transition then by definition there is an arc from  $P_k$  to  $P_i$  in  $Gr$  and if  $k \in (O \setminus O') \cup (I \setminus I') \cup (J \setminus J')$  we are done since (ii) of the hypothesis is violated. If not it must be the case that  $k \notin O \cup I \cup J$  hence it corresponds to a process at an input state and with an empty input queue at the beginning global state  $g$  of the execution sequence. Therefore there must exist a largest index  $n < p$  for which  $receiver(sig(t^n)) = k$  for otherwise  $P_k$  could not have executed  $t^p$  at a later stage without moving away from its input state with an empty queue. Applying the same argument used for  $P_k$  to the process generating  $t^n$  we move backwards in the graph  $Gr$  via intermediate nodes until

we necessarily end up in a process with an index in  $(O \setminus O') \cup (I \setminus I') \cup (J \setminus J')$ . This implies the violation of the hypothesis (ii) of the theorem which proves the desired result when item (2) is violated. The proof of the case where item (3) is violated follows the same logic for the case when (2) is violated hence the details are omitted. *QED*

Now we can give another proof for the read first principle.

**Corollary 1 (Read-first principle)** *Let  $g$  be a global state and let  $P_i$  be a process with the property that its local state  $g[i]$  is an input state where  $g_q[i] \neq \emptyset$  and either  $\text{prior}(g_c[i]) = \emptyset$  or  $\text{prior}(g_c[i]) \cap [g_q[i]] \neq \emptyset$ . Then  $\{t_i\}$  is a persistent set, where  $t_i$  is the unique input transition of  $P_i$ .*

Proof of Corollary :

By definition  $i \in J$ , hence take  $O' = I' = \emptyset$  and  $J' = \{i\}$ . Clearly no path can violate the conditions (i) and (ii) of Theorem 8 since it cannot terminate at a process with index in  $I' \cup K(O') = \emptyset$ . On the other hand (iii) also cannot be violated since  $O' = \emptyset$  which implies the desired result. *QED*

The statement and the proof of Theorem 8 is the guideline for the algorithm that generates a minimal persistent set at a global state  $g$ . After we select an arbitrary process with an index in  $O \cup I$ <sup>2</sup> to initialize we check condition (i) of the Theorem 8. For each path that violates one of the three conditions of Theorem 8 we add the index of the source node to the set  $O' \cup I' \cup J'$ . The algorithm is formally presented below.

### Algorithm 2

**Step 1.** Let  $O' \cup I' = \{j0\}$  for some  $j0 \in O \cup I$  and take  $J' = \emptyset$ ,

**Step 2.** If the condition of Theorem 8 is satisfied, stop. Else go to step 3.

---

<sup>2</sup> As in the previous case starting at an index in  $J$  will terminate the algorithm in a single step by virtue of Theorem 6 or Corollary 1.

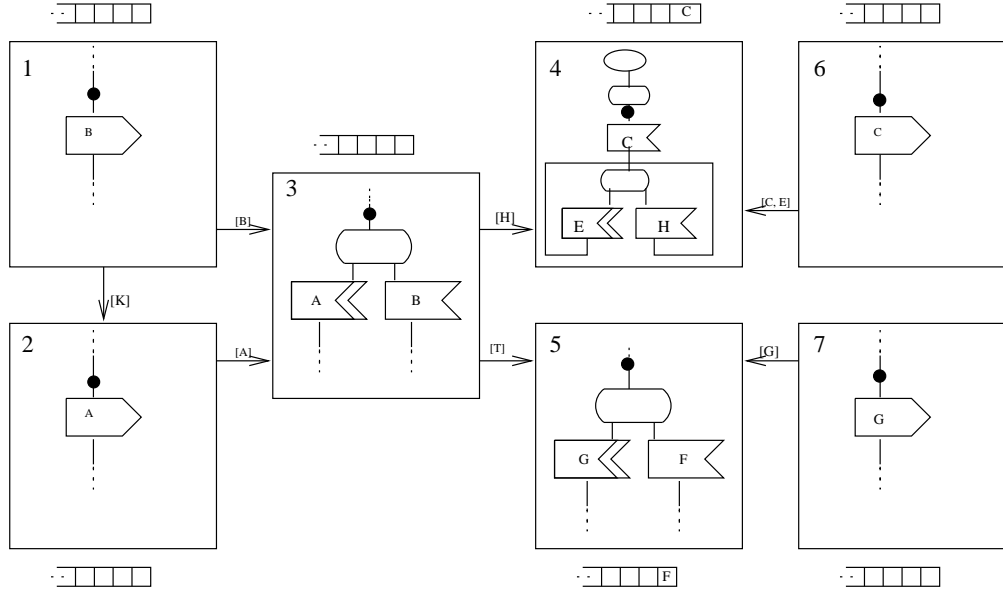


Figure IV.4: An example SDL system

**Step 3.** For each path violating Theorem 8 update  $O'$ ,  $I'$  or  $J'$  depending on the classification of the source node and go to step 2.

Note that the set returned by algorithm satisfies the requirements of Theorem 8 and it has the property that no proper subset of it which includes the initial index  $j_0$  can be proved to be a persistent set using Theorem 8. Therefore the algorithm returns a minimal persistent set in this sense. It is clear that in the worst case all the transitions are dependent with each other, the algorithm reduces to a conventional reachability analysis case where  $A_R$  becomes  $A_G$  and no reduction is obtained. In the best case all the transitions are independent then the algorithm will always select a single element into the persistent set so only one interleaving will be generated for  $n$  transitions instead of  $n!$ .

Figure IV.4 is an example to illustrate the theory developed until now. The present global state of the whole system is the local states denoted by black circles of each process together with queue contents. It is obvious that at the present global state the enabled transition of process 4 which is a signal consumption event is an enabled transition and it is a persistent set. We execute this transition then

to find the persistent set at the next global state we need to use Algorithm 2 above because there is no process which has an enabled priority signal consumption event or a single enabled signal consumption event without any disabled priority signal consumption event at the same state of the process.

Let us begin the first iteration of the algorithm 2 with  $O' = \{7\}$  and  $I' = \emptyset$ . There exist paths starting from processes  $P_1$  and  $P_2$  and ending at  $P_5$  where  $K(O') = \{5\}$  so violating case (i) of Theorem 8 therefore  $O' = O' \cup \{1, 2\} = \{1, 2, 7\}$ . Process 5 is not an element of the process set composed of  $I'$  and  $O'$  and we can find a zero length path starting from this process and ending at this process such that  $7 \in O'$  can send signal  $G$  which is in the priority set of process 5 at this global state, so violating case (iii) of Theorem 8 and  $I' = \{5\}$ . In the second iteration of the algorithm  $K(O') = \{3, 5\}$  but we can not find any processes violating case 1, 2 or 3 of the theorem so we are done and  $P = \{1, 2, 7\} \cup \{5\} = \{1, 2, 5, 7\}$ .

#### IV.4 Further SDL Constructs : Variables and Transition Atomicity

We can deal with an SDL system having variables in two different ways. The first is the method of data flattening. In this approach, we augment each global state with the current values of the variables existing in the system. We also need to add the notion of invisible transition into SSM to be used to model assignment statements and the notion of guarded transitions to be used together with invisible transitions in order to model the decision statements. Then, the results explained until now are still valid. One must consider a process whose next transition is an invisible transition as a  $J$  process when one wants to use Theorem 8. The corollary 1 in this case will also yield another principle, namely invisible first.

The second way of tackling variables is data abstraction. In this approach, we remove all the data related parts from the system and introduce non-determinism into our model to represent decision statements. In other words, we modify the

system as follows: For each assignment statement, we consider as if the statement were not present and the statements above and below of the assignment are directly connected to each other. Each decision box is also removed, and it is thought that, the symbol above the decision box has more than one successor, one for each alternative that the decision box has. Then, we apply our results almost in the same way, except that the definition of  $K(O')$  given in IV.5 is changed accordingly since a process may have more than one outputs enabled at a given state. Also the introduction of NONE input construct of SDL has a little complication on theorems that is: if a process chosen by Algorithm 2 has a NONE input at that state then that NONE transition will also be included in the persistent set. So instead of a process having more than one output transition at the state where decision box is removed, we can have as much NONE transitions at that state as that of the branches of the decision box and following each of these transitions are the transitions in each branch of the decision box. One must notice that, a logical error detected when using this approach does not mean that the original design has the same error. But, if the system after this modification is error free, then the original system is error free.

Each SDL timer has been modeled as an SDL process which sends a timer signal whenever a timer setting signal is received from a process and then non deterministically either sends the timer signal to the process sending the timer setting signal or returns to the initial state if a reset timer signal is received from the process.

Throughout the thesis we assumed the atomicity of input and output statements. It is not difficult to adopt our results to an SDL semantic model in which SDL transitions from an SDL state to another are thought to be atomic. We need to modify the SSM formalism so that an SSM transition can represent an input action followed by (possibly empty) list of output actions. In such a model, we will no further have process of type  $O$  at a global state. All processes are either of type  $I$  or  $J$ . We can still select a subset of processes  $I' \cup J'$  in this case and prove a theorem similar to Theorem 8. The major difference will be the notion of receiver processes ( $K(O')$  in Theorem 8). One must consider all the output

statements present in the transition to form the set of processes that will receive signals by the execution of a transition.

# CHAPTER V

## EXPERIMENTS

We have implemented the theory developed in this thesis in a software tool named POVSDL. This tool can either make a conventional reachability analysis or a reduced reachability analysis using the algorithms developed in the previous chapter. POVSDL is composed of a series of software modules including a parser for input language of POVSDL which implements SSM. We will first explain the POVSDL tool then will come the experimental results with a series of real protocols including ISDN Layer 2 protocol.

### V.1 POVSDL tool

The tool POVSDL (Partial Order Verifier for SDL) was developed to implement the reduction strategies using the methods presented in this thesis, POVSDL also has a conventional reachability analysis option to compare the reduction we can have. An input to POVSDL is a syntactic description of an SSM. The COCKTAIL [GE90, Gro92b] toolbox is used for the scanner and parser stages. The specifications of lexical analyzer(scanner) and parser for the input language of POVSDL are presented in Appendix A and Appendix B respectively. Next we will give some brief information about the scanner and parser tools, after which we will illustrate the software developed.

### V.1.1 Scanner and Parser

COCKTAIL is a compiler construction toolbox that contains a set of tools supporting the construction of almost every phase of a compiler.

The COCKTAIL tools used for POVSDL tool are designed to work in UNIX operating system. These tools are:

REX [Gro92a]                    Lexical analyzer (scanner) generator

LALR [GV92, Gro88]    LALR(1) parser generator

The tools are able to generate modules both in C and Modula-2 programming languages. The generated modules need to be compiled and linked by a C or Modula-2 compiler to generate an executable module.

The COCKTAIL tools have specific input languages. In addition, in almost all the tools semantic actions should be specified either in the implementation language for the generated module, that is C or Modula-2, or in a language specific to the generator tool. The semantic actions may be used to specify any action. The user can make use of the whole power of the implementation language or the specific tool language. For the tools that do not require a specific tool language, the code in the semantic actions is copied unchanged to the generated module with the related pattern replacement text. In other cases, the code in the specific language is converted to the implementation language during generation of the related modules. However, in both cases, the code is not checked by the COCKTAIL tools. The user is responsible for any side-effects that may be caused by the code.

In general, the specification of a component to be input to a COCKTAIL tool starts with a section where pieces of implementation code are given. The code in this section is copied unchanged to specific parts of the generated modules. Statements in the implementation language can be written to specify external, global and local declarations and to initialize and finalize the declared data structures.

REX is a lexical analyzer(scanner) generator. It generates a table-driven scanner that simulates a deterministic finite automaton. Roughly, the input of REX is a set of regular expressions and associated semantic actions which constitute the specification of the scanner to be generated. The output is the source code of

the scanner in the implementation language. In the run-time whenever a string is matched by a regular expression, the code in the associated semantic action is executed.

LALR is an LALR(1) parser generator. The input of LALR is the LALR(1) grammar of the desired language written in BNF or extended BNF possibly with semantic actions written in the implementation language associated with each production. The output is the source code of the parser in the selected implementation language. The generated parsers are table-driven. LALR provides routines for syntax error reporting, repair and recovery for the generated modules.

### V.1.2 POVSDL software

All the generated software modules work in the UNIX operating system. To manage the processing of modules, the UNIX system command *make* is used. *make* reads the specification of what tasks are required to be done from a file called *Makefile*. The *make* command keeps track of the relationships between the modules of the program according to the contents of the *Makefile*. If changes are made in these modules, *make* issues only those commands needed to update the dependent modules so that the changes become effective.

The *Makefile* used for the generation of the lexical analyzer, parser and partial order verifier is given in Figure V.1. The interaction of tools according to the *Makefile* are depicted in Figure V.2.

LALR is invoked with the input file *Parser.lalr* and the following options:

- c : generate C source code
- b : convert ebnf to bnf format
- d : generate definition module (*Parser.h*)
- v : generate debugging information in file *.\_Debug*

As a result the source code for the parser is generated in the implementation language C and stored in files *Parser.c* and *Parser.h*. Information about possible conflicts in the specification is written in file *.\_Debug*.

```

LIB      = /usr/local/cocktail/lib
INCDIR  = /usr/local/cocktail/include
CFLAGS  = -I$(INCDIR)
CC       = gcc -g

povsdl:  main.o Parser.o Scanner.o
         g++ -g $(CFLAGS) Parser.o Scanner.o main.o \
         $(LIB)/libreuse.a -o povsdl

Parser.h Parser.c:  parser.lalr
                  lalr -c -b -d -v parser.lalr;

Scanner.h Scanner.c:  scanner.rex
                    rex -c -d scanner.rex;

main.o:  queue_int.h stack.h Parser.h
         g++ -g $(CFLAGS) -c main.cc

Parser.o:  Parser.h Scanner.h
          $(CC) $(CFLAGS) -c Parser.c

Scanner.o:  Scanner.h
          $(CC) $(CFLAGS) -c Scanner.c

clean:
        rm -f core *.o main

```

Figure V.1: Makefile for the generation of POVSDL

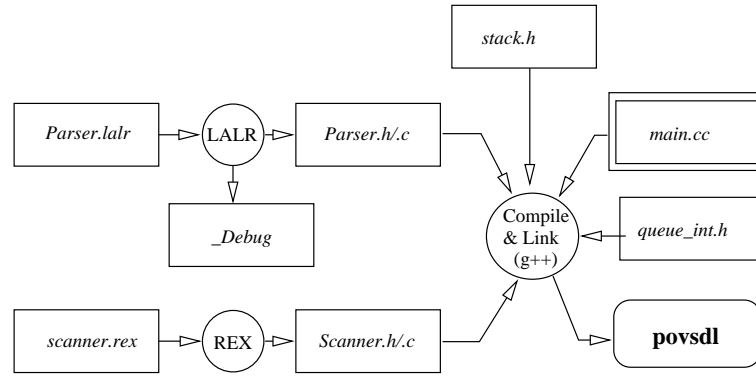


Figure V.2: Interaction of tools to generate the lexical analyzer, parser and POVSDL

The file *scanner.rex* is input to REX with the following options:

- c : generate a lexical analyzer in C
- d : generate definition module for the lexical analyzer

Following these, REX produces the modules *Scanner.h* and *Scanner.c* in the implementation language C.

A main program, named *main.cc*, to drive the parser and to implement verification techniques is written in C++ programming language. The source code for *main.cc* is given in Appendix C. Also files *queue\_int.h* which is an implementation of a bounded queue storing integers (source code is in Appendix D), and *stack.h* which is an implementation of a stack storing the global state of the system (source code is in Appendix E) are written in C++.

To use the generated modules, the files *Scanner.h/c*, *Parser.h/c*, *queue\_int.h*, *stack.h* and *main.cc* are compiled with the GNU C++ compiler “*g++*”. Then the produced object modules are linked to obtain the executable file for partial order verifier, named *povsdl*.

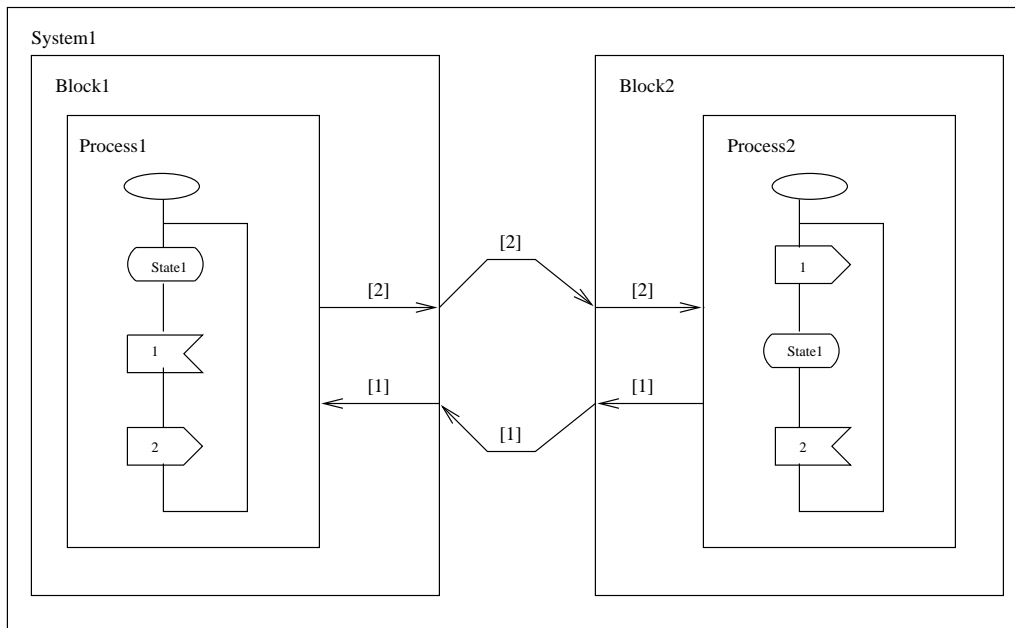
In Figure V.3 there is an example SDL system in graphical form, the corresponding system in input language of POVSDL is also demonstrated. Number of processes in the system are given after the ‘Process\_number’ keyword, process definitions begin with the keyword ‘Process’ and then the process name, the contents of input queues for the processes can be given after ‘Input\_queue’ keyword, states of the processes are denoted by ‘State’ and then the state name, next state

is given after 'Next\_State' keyword, at each state the saved signals are given after the 'Save' keyword, input transitions are denoted by '+' and output transitions by '-' characters, finally a process definition ends with keyword 'End\_process' and name of the process. This is the style of the language parsed by the parser.

POVSDL performs PRA(Persistent Reachability Analysis) according to the algorithm in Figure IV.1 of chapter IV. Also CRA(Conventional Reachability Analysis) is performed by replacing function *Persistent\_Set(s)* in the algorithm by *enabled\_set(s)* (i.e., all enabled transitions at the global state  $s$ ). These are two options of the tool. The user is also asked to give a bound for the input queues of the processes. During verification the visited states are stored in a hash table and about one hundred bytes are needed to store a single state with a queue length of 20 (a state is a composition of each process local state and each process queue contents). A simple hash function is used for generating a hash value then a linked list style hash table is used. In POVSDL the states are explicitly enumerated for verification with no collision risk as opposed to using a supertrace style approach [Hol91] that may further reduce the state space generated.

The main function in *main.cc* is *exhaust* which takes the initial state and the parsed protocol structure as parameters and implements the algorithm in Figure IV.1 of chapter IV. Inside this function stack and hash tables are initialized and a depth first search is performed throughout which you can call function *enable* or function *persistent* for CRA and PRA respectively. Then according to the  $T$  returned by the functions (i.e., set of all enabled transitions at that global state or set of transitions returned by the functions implementing Algorithm 2 of chapter IV for finding the persistent set at that global state) the next global states are found and pushed onto stack. Then the states are popped and again the next global states are found and pushed onto stack. This step continues until the stack becomes empty that is when there are no new next states to generate. Then a table is displayed showing the amount of time elapsed, the number of transitions, the number of states, the number of deadlocks and the number of unspecified receptions.

Function *enable* finds the set of enabled transitions at that global state of



Example SDL system

Process_number 2	
Process 1	Process 2
Input_queue	Input_queue
Transition_number 2	Transition_number 2
State 1	State 1
Save	Save
+ 1	- 1 toProcess 1
Next_State 2	Next_State 2
State 2	State 2
Save	Save
- 2 toProcess 2	+ 2
Next_State 1	Next_State 1
End_process 1	End_process 2

The above system in the input language of POVSDL

Figure V.3: Graphical representation of an example SDL system and the corresponding compiled system for POVSDL tool input

the system which it takes as a parameter by making use of the function called *enable\_s* which takes control state of a single process, the process name, and the protocol structure as parameters and returns the enabled transition at that state of a process in a structure.

Function *persistent* takes present global state and the protocol structure as input parameters. This function checks whether we can make use of *read first principle* if yes then the transition is returned in a structure if not then a function called *algo5* is used which implements Algorithm 2 of chapter IV to find a persistent set.

## V.2 Experimental results

We have worked on several realistic examples to illustrate the results of our reduction methodologies. The methodology followed in verification of these protocols are as follows. First the protocols are specified in SDL, second as explained in the previous chapter the data parts of the protocols are excluded so that translation into SSM is direct, however while doing exclusion of data part the system then should be able to generate the paths that may be followed by the values of data. This is captured by letting the SSM specification make nondeterministic transitions as explained at the end of the previous chapter. The sample protocols used are as follows.

- ABP is the alternating bit protocol with 5 processes.
- SSSM93 is a connection establishment protocol taken from [NIUS93]. This protocol sets up and releases a call. There are 2 processes exchanging signals and an environment sending and receiving signals to and from processes.
- SSM95 and pro10 are protocols which we generated for experimental purposes and they simply exchange signals in an arbitrary manner.
- ISDN, ISDN (Integrated Services Digital Network) Layer 2 (Data Link Layer) provides a secure, error-free connection between two end-points connected by a physical medium, responsible for detecting and retransmitting

lost frames. The method for recovering from a lost frame is based on the expiration of a timer. A timer is started every time a command frame is transmitted and is stopped when the appropriate response is received. The SDL representation of point-to-point procedures of the data link layer given in ETSI [ETS91] recommendations has formed the backbone of the Layer 2 SDL specification in the study. The most complicated process in specification provides both acknowledged and unacknowledged information transfer services. Acknowledged information transfer also implements the services, link establishment, link re-establishment and link release. Another process performs TEI (Terminal Endpoint Identifier) assignment, check and removal procedures. The protocol has been specified in SSM with exclusion of data part as explained above. The input code for the POVSDL tool for ISDN example is in the Appendix F.

Experiments were performed by using two methods.

- CRA conventional reachability analysis is exhaustive verification with depth first search order.
- PRA persistent reachability analysis is the partial order methodology which we developed in this thesis and uses Algorithm 2 of previous chapter and the read-first principle.

The POVSDL tool checks for deadlocks and unspecified receptions. Table V.1 presents for each of the methods and protocols number of visited states and number of transitions. The user is able to put a bound on the input queue lengths of processes in the protocols which is practical.

The reductions obtained are of order of magnitude. It is obvious that in all of the protocols there is a huge amount of reduction both in terms of states and transitions generated compared to conventional reachability analysis case. The PRA method is slower than CRA due to the computations done for calculating persistent sets. It can be observed from the ISDN example that complex protocols can also be verified easily using the PRA method which is not the case

Table V.1: Results of verification

Protocol	Algorithm	States	Transitions
ABP	CRA	12223	24580
	PRA	95	97
SSM93	CRA	132224	484265
	PRA	55	70
SSM95	CRA	3227	12585
	PRA	105	142
pro10	CRA	63	135
	PRA	15	14
ISDN	CRA	76949	351190
	PRA	1545	1892

for commercial verifiers because there is no such partial order methodology used in commercial market. They use exhaustive verification as in CRA method or a supertrace style verification which is not exhaustive due to the method followed in storing the visited states. It is quite clear that PRA method developed is the most efficient verification engine developed until now for SDL systems.

Partial order methods bring no risk for the state space search of the systems analyzed, but they can yield quite nice improvements in the performance of verification. By the use of partial order methods, more complex systems can be verified.

The reduction obtained depends on the coupling between the processes in the system. When coupling is very tight, partial-order methods yield no reduction, and the PRA method can reduce to CRA method. But when the coupling is very loose, the reduction can be very impressive.

# CHAPTER VI

## CONCLUSION

### VI.1 Summary

Partial order methods take benefit from modelling of concurrency by interleaving by which all interleavings of all concurrent transitions of a system are represented in its state space. It is shown that exploring all these states is not necessary for verification. The algorithms we develop search only a reduced part of the state space of a concurrent system that is sufficient for checking deadlock properties of a system. We have presented the notion of independency of transitions and equivalent traces which lead to selecting a subset of all interleavings and explain the transitions that are independent in SDL systems. In order to select the subset of all interleavings of a system a definition called persistent sets is given which selects only a subset of all transitions which are enabled at a global state of a system. We then give algorithms to find persistent sets both for the basic SDL subset and then the extended SDL case which guarantee deadlock detection. Application of the methods for further constructs of SDL are shown. A verification engine (POVSDL) implementing the ideas presented in this thesis is explained and experiment results for various real protocols are given.

An immense amount of reduction has been obtained on systems which can make good use of read-first methodology. It is clear from the experimental results that the theory developed in this thesis is the most efficient verification method

developed for SDL systems. There is not any work comparable to our study in the literature and the ones done cover a smaller subset of SDL than we cover.

## VI.2 Future Work

We are planning to introduce data in SDL into SSM model. In this thesis methodologies for deadlock detection have been developed, this work can be extended for verification of liveness properties as well. A compiler from SDL to SSM will automate the translation step and it will enable us to plug POVSDL into various commercial SDL tools. A more efficient code for POVSDL that will make better use of time and space can be written. This work has been a model for the SDL-COSPAN compiler work and the ideas are used in the compiler being developed.

Our work will develop in a way to understand the mechanisms in generating states during verification. The problem is that with all [God95], [Hol91] state space verification techniques, at a global state we skip some other next states that are generated during conventional reachability analysis which gives the improvement in all the methods. But it is unknown whether we will always come to these skipped states and generate them at a later phase of the verification.

Further complexity reduction improvements are possible via : (i) implementation of a collision risk technique similar to that utilized by Holzmann [Hol91], (ii) combining leaping reachability analysis [OU94] with the persistent set idea by generating a collection of persistent sets at each global state and taking a multi-step forward leap by executing one transition from each set synchronously for each combination of individual selections. For example, any number of non-output transitions enabled at a state, can be executed simultaneously. We can derive such synchronously executable sets of enabled transitions including output statements also.

## REFERENCES

- [CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, pages 8(2):244–263, January 1986.
- [ETS91] *ETSI, ETS 300 125, "Integrated Services Digital Network(ISDN)"; User-Network Interface Data Link Layer Specification Application of CCITT Recommendations Q.920/I.440 and Q921/I.441*, September 1991.
- [FGM<sup>+</sup>92] J.C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of lotos programs. *Proc. of the 14th International Conference on Software Engineering ICSE'92*, May 1992.
- [FO93] O. Færgemand and A. Olsen. *New Features in SDL-92, SDL Newsletter, No: 16, pp.10-29*, 1993.
- [Gai88] H. Gaifman. Modeling concurrency by partial orders and nonlinear transition systems. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS 354*, pages 467–488, 1988.
- [GE90] J. Grosch and H. Emmelmann. A tool box for compiler construction. Compiler Generation Report 20, GMD Forschungsstelle an der Universität Karlsruhe, 1990.
- [GEO96] *ObjectGEODE*. Verilog SA, 1996.
- [Gla95] A. Glaser. *S/R Reference Manual*. AT&T Bell Laboratories, June 1995.
- [God95] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State Explosion Problem*. PhD thesis, Universite De Liege, 1994-95.
- [Gro88] J. Grosch. Lalr - a generator for efficient parsers. Compiler Generation Report 10, GMD Forschungsstelle an der Universität Karlsruhe, 1988.

- [Gro92a] J. Grosch. Rex - a scanner generator. Compiler Generation Report 5, GMD Forschungsstelle an der Universität Karlsruhe, 1992.
- [Gro92b] J. Grosch. Toolbox introduction. Compiler Generation Report 25, GMD Forschungsstelle an der Universität Karlsruhe, 1992.
- [GV92] J. Grosch and B. Vielsack. The parser generators lalr and ell. Compiler Generation Report 8, GMD Forschungsstelle an der Universität Karlsruhe, 1992.
- [Hol91] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
- [KP92] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6:107–120, 1992.
- [Kur94] R. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
- [LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. *Proc. of the 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, January 1985.
- [Maz89] A. Mazurkiewicz. Basic notions of trace theory. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS 354*. Springer Verlag, 1989.
- [NIUS93] F. Nitta, A. Ito, E. Utsunomiya, and H. Saito. Protocol validation for specifications in SDL. In *SDL'93 Using Objects*, pages 193–204. North-Holland, 1993.
- [OFMP<sup>+</sup>94] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J.R.W. Smith. *Systems Engineering Using SDL-92*. North-Holland, 1994.
- [OU94] K. Özdemir and H. Ural. Deadlock detection in CFSM models via simultaneously executable sets. In *6th Int. Conf. On Communication and Information*, pages 673–688, Peterborough, Ontario, Canada, 1994.
- [Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proc. Of 6th Conference on Computer Aided Verification*, pages 377–390, Stanford, June 1994. Springer-Verlag. LNCS 818.
- [Pra86] V. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

- [QS91] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in caesar. *Proc. of the 5th Intl. Symp. on Programming*, pages 337–351, 1991.
- [SDL92] *Functional Specification and Description Language (SDL), CCITT Blue Book, Recommendation Z.100*. Geneva, 1992.
- [SDT95] *SDT 3.02*. Telelogic AB, 1995.
- [SY97] A. Şen and H. Yenigün. Verification of SDL with persistent sets. Technical report, SRDC, TÜBİTAK, Ankara, Turkey, January 1997.
- [TGH95] D. Toggweiler, J. Grabowski, and D. Hogrefe. Partial order simulation of SDL specifications. In *SDL '95 with MSC in Case*, pages 293–306. Elsevier, 1995.
- [Tog95] D. Toggweiler. *Efficient Test Generation for Distributed Systems Specified by Automata*. PhD thesis, University of Berne, May 1995.
- [Val90] A. Valmari. A stubborn attack on state explosion. In *Computer Aided Verification*, pages 156–165, New Brunswick, NJ, USA, June, 1990.
- [VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, June 1986.
- [WH93] M. Woodman and B. Heal. *Introduction to VDM*. McGraw-Hill, 1993.

# APPENDIX A

## POVSDL SCANNER SPECIFICATION

```
EXPORT{
#include "Positions.h"
#include "StringMem.h"
#include "Idents.h"
# if defined __STDC__ | defined __cplusplus
# define ARGS(parameters)      parameters
# else
# define ARGS(parameters)      ()
# endif

typedef struct { tPosition yyPos; int Integer; } yyintConst;
typedef struct { tPosition yyPos; tIdent Name; } yyname;
typedef struct { tPosition yyPos; tStringRef String; } yycharacterString;

typedef struct {
    tPosition Position;
    yyintConst intConst;
    yyname name;
    yycharacterString characterString;
    char* string;
} tScanAttribute;

extern void ErrorAttribute ARGS((int Token, tScanAttribute * Attribute));
}

GLOBAL{
#include <math.h>
#include "Memory.h"
#include "StringMem.h"
#include "Idents.h"
#include "Errors.h"

void ErrorAttribute
# if defined __STDC__ | defined __cplusplus
(int Token, tScanAttribute * Attribute)
# else
(Token, Attribute) int Token; tScanAttribute * Attribute;
# endif
{}
}

LOCAL {
static char Word [10][10][64];
static char Word1 [64];
```

```

static char Word2 [600][64];
static char Word3 [600][64];
static char Word4 [600][64];
static char Word5 [600][64];
static char Word6 [600][64];
static char Word7 [600][64];
static char Word8 [600][10][64];
static char Word9 [600][64];
static int val=0;
static int val1=0;
static int val2=0;
static int val3=0;
static int val4=0;
static int val5=0;
static int val6=0;
static int val7=0;
static int val8=0;
static int val9=0;
static int val10=0;
}

DEFAULT {
MessageI("illegal character:", xxError,Attribute.Position,xxCharacter,TokenPtr);
}

BEGIN{
extern char* myfile;
BeginFile(myfile);
printf("File %s is opened\n",myfile);
}

CLOSE{
extern char* myfile;
CloseFile(myfile);
printf("File %s is closed\n",myfile);
}

DEFINE
digit      = {0-9}      .
letter     = {a-z A-Z}  .
nl         = {\n}      .

START st1, st2 , st3, st4, st5, st6, st7, st8, st9, st10, st11, st12, no

RULES
#STD# "Process_number" : { yyStart(st1); return 1; }
#STD# "Process"        : { yyStart(st2); return 2; }
#STD# "End_process"    : { yyStart(no); return 3; }
#STD# "Input_queue"    : { val=val+1; yyStart(st3); return 4; }
#STD# "Transition_number" : { yyStart(st1); return 5; }
#STD# "State"          : { yyStart(st4); return 6; }
#STD# "Next_State"     : { yyStart(st5); return 7; }
#STD# "-"              : { yyStart(st6); return 8; }
#STD# "Save"           : { val7=val7+1; yyStart(st9); return 14; }
#STD# "NONE"          : { return 13; }
#STD# "+"              : { yyStart(st7); return 9; }
#STD# "priority"      : { yyStart(st10); return 15; }
#STD# "toProcess"     : { yyStart(st8); return 11; }

#st1# digit+          : { (void) GetWord(Word1);
                        Attribute.intConst.Integer = atoi (Word1);
                        yyStart(STD); return 10; }

#st2# digit+          : { (void) GetWord(Word2[val1]);
                        Attribute.intConst.Integer = atoi(Word2[val1]);
                        val1=val1+1; yyStart(STD); return 10; }

```

```

#st3# (digit+ | nl)      : { (void) GetWord(Word[val][val10]);
                          if (*Word[val][val10]=='\n') { yyStart(STD); }
                          else
                          {Attribute.intConst.Integer=atoi(Word[val][val10]);
                           yyStart(st12); val10=val10+1; return 10; }; }

#st4# digit+            : { (void) GetWord(Word3[val2]);
                          Attribute.intConst.Integer = atoi(Word3[val2]);
                          val2=val2+1; yyStart(STD); return 10; }

#st5# digit+            : { (void) GetWord(Word4[val3]);
                          Attribute.intConst.Integer = atoi(Word4[val3]);
                          val3=val3+1; yyStart(STD); return 10; }

#st6# digit+            : { (void) GetWord(Word5[val4]);
                          Attribute.intConst.Integer= atoi(Word5[val4]);
                          val4=val4+1; yyStart(STD); return 10; }

#st7# digit+            : { (void) GetWord(Word6[val5]);
                          Attribute.intConst.Integer= atoi(Word6[val5]);
                          val5=val5+1; yyStart(STD); return 10; }

#st8# digit+            : { (void) GetWord(Word7[val6]);
                          Attribute.intConst.Integer = atoi(Word7[val6]);
                          val6=val6+1; yyStart(STD); return 10; }

#st9# (digit+ | nl)     : { (void) GetWord(Word8[val7][val9]);
                          if (*Word8[val7][val9]=='\n') { yyStart(STD); }
                          else
                          {Attribute.intConst.Integer=atoi(Word8[val7][val9]);
                           yyStart(st11); val9=val9+1; return 10; }; }

#st10# digit+           : { (void) GetWord(Word9[val8]);
                          Attribute.intConst.Integer = atoi(Word9[val8]);
                          val8=val8+1; yyStart(STD); return 10; }

#st11# "$"              : { yyStart(st9); return 16; }

#st12# "$"              : { yyStart(st3); return 16; }

#no# digit+             : -{ yyStart(STD); return 10;}

```

# APPENDIX B

## POVSDL PARSER SPECIFICATION

```
PARSER

GLOBAL{
#include "StringMem.h"
#include "Idents.h"
typedef union {
tScanAttribute Scan;
} tParsAttribute;

int Digit;
int pars_proc_name[100];
int pars_queue[10][10];
int pars_states[600];
int pars_next_states[600];
int pars_transno[100];
char pars_sig_type[600];
int pars_signal[600];
int pars_toprocess[600];
int pars_save[600][10];
int no1; int no2; int no3; int no4; int no5; int no6;
int no7; int no8; int no9;
}

BEGIN { BeginScanner ();
no1=-1; no2=-1; no3=0; no4=-1; no5=-1; no6=-1; no7=0;
no8=-1; no9=-1;
}

CLOSE {
free(pars_queue);
free(pars_sig_type);
free(pars_signal);
free(pars_save);
}

TOKEN
'Process_number' = 1
'Process' = 2
'End_process' = 3
'Input_queue' = 4
'Transition_number' = 5
'State' = 6
'Next_State' = 7
',' = 8
```

```

'+'           = 9
digit        = 10
'toProcess'  = 11
'NONE'      = 13
'Save'      = 14
'priority'  = 15
'$'         = 16

RULE

system       : process_no processes.

process_no   : 'Process_number' digit
              { Digit = $2.Scan.intConst.Integer; } .

processes    : processesy ( processesy )* .

processesy   : process_name process_structure end_process_name .

process_name : 'Process' digit
              { no2 = no2 + 1;
                pars_proc_name[no2] = $2.Scan.intConst.Integer; } .

end_process_name : 'End_process' digit.

process_structure : queue infra .

queue        : 'Input_queue' q1
              { no3 = no3 + 1; no8=-1; } .

q1           : ( q2 '$' )* .

q2           : digit
              { no8 = no8 + 1;
                pars_queue[no3][no8]=$1.Scan.intConst.Integer; }.

infra       : transs states .

transs      : 'Transition_number' digit
              { no1 = no1 + 1;
                pars_transno[no1] = $2.Scan.intConst.Integer; }.

states      : statesy ( statesy ) * .

statesy     : statesyf transition statesyl .

statesyf    : 'State' digit
              { no4 = no4 + 1;
                pars_states[no4]=$2.Scan.intConst.Integer; }.

statesyl    : 'Next_State' digit
              { no5 = no5 + 1;
                pars_next_states[no5]=$2.Scan.intConst.Integer; }.

transition  : save_list (input_transition | output_transition |
                       none_transition | priority_transition ).

save_list   : 'Save' s1
              { no7 = no7 + 1; no9 = -1; } .

s1          : ( s2 '$' )* .

s2          : digit
              { no9 = no9 + 1;
                pars_save[no7][no9]=$1.Scan.intConst.Integer; }.

input_transition : '+' digit
                 { no6=no6+1;
                   pars_sig_type[no6]='+';

```

```

        pars_signal[no6]=$2.Scan.intConst.Integer;
        pars_toprocess[no6]=999; }.

output_transition : '-' digit 'toProcess' digit
{ no6=no6+1;
  pars_sig_type[no6]='-';
  pars_signal[no6]=$2.Scan.intConst.Integer;
  pars_toprocess[no6]=$4.Scan.intConst.Integer; }.

none_transition  : 'NONE'
{ no6=no6+1;
  pars_sig_type[no6]='N';
  pars_signal[no6]=0;
  pars_toprocess[no6]=999; }.

priority_transition: 'priority' digit
{ no6=no6+1;
  pars_sig_type[no6]='p';
  pars_signal[no6]=$2.Scan.intConst.Integer;
  pars_toprocess[no6]=999; }.

```

# APPENDIX C

## MAIN.CC SOFTWARE CODE

```
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <sys/time.h>
#include <string.h>
#include <sys/resource.h>

#include "queue_int.h" //bounded integer array queue
#include "stack.h" //stack which stores
//global system state

#define STACKSIZE 200000
#define HASHSIZE 30000

char* myfile;
int sel;
int lenq;
int en =0;
int pr =0;
int lnk=0;
int stn=0;
int gq=0;
int relno=0;
int hnum[HASHSIZE];
int maxh;

extern "C" {
int Parser(); void CloseParser ();
int Digit;
int pars_proc_name[10];
int pars_transno[600];
int pars_queue[10][10];
int pars_save[600][10];
int pars_states[600];
int pars_next_states[600];
char pars_sig_type[600];
int pars_signal[600];
int pars_toprocess[600];
};

/*****
Initial menu asking for the name of the input protocol
which will be verified
*****/
```

```

char* initialize1(){
char * file_name= new char[40];
system("clear");
cout <<"====VERIFICATION=====" << endl;
cout <<"|                               |" << endl;
cout <<"| 0. Exit                          |" << endl;
cout <<"|                               |" << endl;
cout <<"| 1. Enter a file name              |" << endl;
cout <<"|                               |" << endl;
cout <<"=====" << endl;

cout << "Enter your selection: " ;
cin >> sel;

if (sel==0) exit(1);
if (sel==1) {
    cout << "Filename: ";
    cin >> file_name;
};
return(file_name);

};

/*****
Algorithms that the protocol can be verified
*****/

int* initialize2(){
system("clear");
cout<<"====PERSISTENT/EXECUTE=====" << endl;
cout<<"|                               |" << endl;
cout<<"| 2. Conventional Reachability Analysis |" << endl;
cout<<"|                               |" << endl;
cout<<"| 4. Persistent Reachability Analysis |" << endl;
cout<<"|                               |" << endl;
cout<<"| 5. Exit                          |" << endl;
cout<<"|                               |" << endl;
cout<<"=====" << endl;

cout << "Enter your selection: " ;
cin >> sel;

if ((sel==0) || (sel==1) || (sel==2) || (sel==3) || (sel==4) )
{
    int* global_state=new int[Digit];
    for ( int j=1; j<=Digit; j++)
    {
        cout << "Enter the global state: ";
        cin >> global_state[j-1];

    };
    return(global_state);
};
if (sel==5) exit(1);
};

/*****
Function to find the number of signals
in the input queue of a process
*****/

int length_of_queue(int* queue)
{
int g;
for( g=1; g<=lenq;g++ )
{
if ( queue[g-1] == 0 ) break;
};
};

```

```

    return (g-1);
};

/*****
Checks whether a signal is in the savelist
of a state of a process
*****/

int isinSave(int s, int* savelist)
{
    int ret=0;
    for(int g=1; g<=length_of_queue(savelist); g++)
    {
        if ( savelist[g-1] == s )
            { ret=g ; break;};
    }
    return(ret);
};

/*****
Structure to store the informations about
a process's state
*****/

class state {
public:
    state(int, int, char, int, int, int* );
    state();
    int state_name;
    char inout;
    int signal;
    int toprocess;
    int next_state_name;
    int* save;
};

state::state(int myname,
             int my_next_state_name,
             char mytpe,
             int mysignal,
             int processto,
             int* mysave)
{
    state_name = myname;
    next_state_name = my_next_state_name;
    signal=mysignal;
    save=new int[length_of_queue(mysave)];
    save=mysave;
    toprocess=processto;
    inout=mytpe;
};

state::state()
{};

/*****
Structure to store the informations about
a process's input queue, transition
numbers and states
*****/

class processes {
public:
    processes(int, int, int*, int );
    processes();
    int process_name;
    int_queue* input_queue;
    int transition_number;
    state* mystates;
};

```

```

};

processes::processes(int myname,
                    int mytransno,
                    int* myqueue,
                    int st)
{
    process_name = myname;
    transition_number = mytransno;
    input_queue= new int_queue(lenq);
    for(int t=1; t<=length_of_queue(myqueue); t++)
    {
        input_queue->enqueue(myqueue[t-1]);
    };
    mystates=new state[mytransno];
    for(int i=1;i<=mytransno;i++)
    {
        mystates[i-1]=state(pars_states[st+i-1],
                            pars_next_states[st+i-1],
                            pars_sig_type[st+i-1],
                            pars_signal[st+i-1],
                            pars_toprocess[st+i-1],
                            pars_save[st+i-1]
                            );
    }
};

processes::processes()
{};

/*****
Main structure to store information
about the protocol to be verified
(i.e. creates objects for all
constructs in the test file)
*****/
class menu{
public:
    menu();
    processes* myprocess;
};

menu::menu(){
    Parser();
    myprocess=new processes[Digit];
    int k=0;
    for(int i=1;i<=Digit;i++)
    {
        myprocess[i-1]= processes(pars_proc_name[i-1],
                                pars_transno[i-1],
                                pars_queue[i-1],k);

        k=k+pars_transno[i-1];
    }
};

/*****
Structure to store the set of processes
returned for persistent set selection
*****/
struct rel{
    int* ars;
    int len ;
    rel(){ ars=new int[Digit];len=0; relno++;};

    rel(int as) { ars=new int[Digit]; relno++;
                ars[0]= as; len=1; };
    ~rel(){ delete [] ars; relno--;};
};

```

```

struct link {
    link *next;
    mytype* a;
    link() { next = 0; lnk++; };
    ~link() { delete a; delete next; lnk--; };
};

rel* table;
rel* tablep;
link** H;

rel* relation(int process, menu client)
{
    rel* myrel= new rel;
    int val=0;
    int k, h, u;
    for( u=1; u<=Digit; u++)
    {
        for( h=1; h<=pars_transno[u-1];h++)
        {
            if(client.myprocess[u-1].mystates[h-1].toprocess ==
                process)
            {
                val =0;
                for( k=1; k<=myrel->len;k++)
                {
                    if(myrel->ars[k-1] ==
                        client.myprocess[u-1].process_name) val=1;
                };
                if ( val == 0)
                {
                    myrel->ars[myrel->len] = client.myprocess[u-1].process_name;
                    myrel->len++ ;
                };
            };
        };
    };
    return(myrel);
};

int expo(int y, int h)
{
    int r=1;
    for (int j=1; j<=h;j++)
    { r=r*y; };
    return(r);
};

/*****
Function to find the union of
two sets of type rel
*****/

rel* unions(rel* set1, rel* set2)
{
    int val=0;
    int element, t;
    for( t=1; t<=set2->len; t++)
    {
        element= set2->ars[t-1];
        val=0;
        for(int y=1; y<=set1->len; y++)
        { if(set1->ars[y-1]==element ) val =1; };
        if( val == 0)
        {
            set1->ars[set1->len]= element;
            set1->len=set1->len+1;
        };
    };
};

```

```

};
return(set1);
};

/*****
Function to find whether a set of
type rel is a subset of another
set of type rel
*****/

int subset(rel* set2, rel* set1)
{
    int val=0;
    int m=1;
    int element, t, y;
    for( t=1; t<=set2->len; t++)
    {
        element= set2->ars[t-1];
        val=0;
        for( y=1; y<=set1->len; y++)
        {
            if(set1->ars[y-1]== element ) val =1;
        };
        if( val == 0) m=0 ;
    };
    return m;
};

void recurse(rel* prime,int i, menu client)
{
    rel* dprime= new rel;
    rel* my1=new rel;
    int g;
    for( g=1; g<=prime->len; g++)
    {
        my1=relation(prime->ars[g-1], client);
        dprime=unions(dprime, my1);
    };

    if(!subset(dprime, &table[i]))
    {
        table[i]=*unions(&table[i],dprime);
        recurse(dprime, i, client);
    };
    delete my1;
    delete dprime;
};

void relationstar(menu client)
{
    rel* my=new rel;
    int length_of_set, kr, fg;
    for(int u=1; u<=Digit; u++)
    {
        my=relation(client.myprocess[u-1].process_name, client);
        length_of_set=my->len;
        table[u-1]=*my;
        tablep[u-1]=*my;
        recurse(&table[u-1],u-1, client);
        kr=0;
        for( fg=1; fg<=table[u-1].len; fg++)
        {
            if(( client.myprocess[u-1].process_name ==
                table[u-1].ars[fg-1] ) && (kr==0)) kr=1;
            if((kr==1)&& (fg!=table[u-1].len))
                table[u-1].ars[fg-1]=table[u-1].ars[fg];
        };
        if (kr==1) table[u-1].len--;
    };
};

```

```

delete my;
};

int converter(menu client, int pro_name)
{
for(int j=1; j<=Digit; j++)
{
if(client.myprocess[j-1].process_name== pro_name){break;};
};
return (j-1);
};

/*****
Structure to store the set of enabled
transitions
*****/

struct enablest{
int length_of_enabled;
state* enabled;
processes* proc_set;
int* enabled_by_save;
int* priority_not_enabled;
enablest(){
enabled = new state[10];
proc_set=new processes[10];
enabled_by_save = new int[10];
priority_not_enabled = new int[10];
length_of_enabled=0;
en= en + 1;
pr = pr + 1; };
~enablest(){
delete [] enabled;
delete [] proc_set;
delete [] enabled_by_save;
delete [] priority_not_enabled;
en = en - 1;
pr = pr -1; };
};

/*****
Function which finds enabled transition
of a process at the given control state
*****/
enablest* enable_s(int each_state, int proc, menu client)
{
int len =0;
int u, t;
int there_is_priority_input=0;
int another_check =0;
enablest* rts=new enablest;
for( t=1; t<=pars_transno[proc];t++)
{

if(client.myprocess[proc].mystates[t-1].state_name ==
each_state )
{
//IF the signal is an output signal
if (client.myprocess[proc].mystates[t-1].inout == '-')
{
for( u=1; u<=Digit;u++)
{
if(client.myprocess[proc].mystates[t-1].toprocess==
client.myprocess[u-1].process_name )
{break; };
};
if(client.myprocess[u-1].input_queue->mynumb < (lenq))
{
rts->enabled[len]=client.myprocess[proc].mystates[t-1];

```

```

rts->proc_set[len]=client.myprocess[proc];
rts->enabled_by_save[len]=0;
len++;
};
};
        //IF the signal is an input signal
if(
( client.myprocess[proc].mystates[t-1].inout == '+' ) &&
( !( client.myprocess[proc].input_queue->emptyQ() ) ) &&
( there_is_priority_input == 0 )
)
{
        //IF the signal is at the head of the queue
if(
client.myprocess[proc].input_queue->headQ() ==
client.myprocess[proc].mystates[t-1].signal
)
{
rts->enabled[len]= client.myprocess[proc].mystates[t-1] ;
rts->proc_set[len]=client.myprocess[proc];
rts->enabled_by_save[len]=0;
len++;
};
        //IF the signal is after a saved signal in the queue
if(
client.myprocess[proc].input_queue->saveQ(client.myprocess[proc].mystates[t-1].signal,
client.myprocess[proc].mystates[t-1].save)
)
{
rts->enabled[len]= client.myprocess[proc].mystates[t-1] ;
rts->proc_set[len]=client.myprocess[proc];
rts->enabled_by_save[len]=1;
len++;
};
};
        //IF the signal is a priority input signal
if(
(client.myprocess[proc].mystates[t-1].inout == 'p') &&
(client.myprocess[proc].input_queue->isinQ(client.myprocess[proc].mystates[t-1].signal))
)
{
len =0;
rts->enabled[len]= client.myprocess[proc].mystates[t-1] ;
rts->proc_set[len]=client.myprocess[proc];
rts->enabled_by_save[len]=0;
len++;
there_is_priority_input = 1;
};

if(
(client.myprocess[proc].mystates[t-1].inout == 'p') &&
(!client.myprocess[proc].input_queue->isinQ(client.myprocess[proc].mystates[t-1].signal))
)
{ another_check=1; };

if(sel!=3 && sel!=4 && sel!=0)
{
if (client.myprocess[proc].mystates[t-1].inout == 'N')
{
rts->enabled[len]= client.myprocess[proc].mystates[t-1] ;
rts->proc_set[len]=client.myprocess[proc];
len++;
};
};
};
};

if (another_check == 1 )
rts->priority_not_enabled[len-1] = another_check;

```

```

    rts->length_of_enabled = len;
    delete rts;
    return(rts);
};

/*****
Function which finds all enabled transitions
at a global state of the system
*****/

enablest* enable(int* present_state, menu client)
{
    int length_of_enabled_set =0;
    enablest* rts=new enablest;
    enablest* retr;
    int each_state, h;
    for ( int y=1; y<=Digit; y++ )
    {
        each_state = present_state[y-1];
        retr=enable_s(each_state, y-1, client);
        for( h=1; h<=retr->length_of_enabled;h++)
        {
            rts->enabled[length_of_enabled_set] = retr->enabled[h-1] ;
            rts->proc_set[length_of_enabled_set] = retr->proc_set[h-1];
            rts->enabled_by_save[length_of_enabled_set] = retr->enabled_by_save[h-1];
            rts->priority_not_enabled[length_of_enabled_set] = retr->priority_not_enabled[h-1];
            length_of_enabled_set++;
        }
    };
    rts->length_of_enabled=length_of_enabled_set;
    return(rts);
};

/*****
Function which prints the screen
information about the transitions inputted in terms
of state name, transition name, next state name etc.
*****/

void print_to_scr(state* enabled_set,
                  processes* pro_set,
                  int length_of_enabled_set)
{
    for (int r=1; r<=length_of_enabled_set; r++)
    {
        cout << r-1 << " ";
        cout << "Process: " << pro_set[r-1].process_name << " ";
        cout << "State: " <<enabled_set[r-1].state_name << " ";
        cout << "Transition: " <<enabled_set[r-1].inout << " ";
        cout << enabled_set[r-1].signal << " ";
        cout << "To: " <<enabled_set[r-1].toprocess << " ";
        cout << "Next_State: " <<enabled_set[r-1].next_state_name << " " << endl;
    };
};

int isina(rel* a, int al)
{
    int k=0;
    for(int h=1; h<=a->len; h++)
    {
        if(a->ars[h-1]== al) k=1;
    };
    return(k);
};

rel* recurse2(rel* a, int * global, int proc, menu client)
{
    rel* g=new rel;
    enablest* fty;

```

```

int ftx;
int al, hf, each, gu, kn;
for( kn=1; kn<=tablep[proc].len; kn++)
{
    al=tablep[proc].ars[kn-1];
    hf=converter(client, al);
    if(!isina(a, al))
    {
        each= global[hf];
        fty=enable_s(each, hf, client);
        ftx = fty->length_of_enabled;
        rel* ak=new rel(al);
        delete ak;
        if (ftx!=0)
            g=unions(g, ak);
        else{
            rel* f=recurse2(unions(a, ak), global, hf, client);
            g=unions(g, f);
            delete f;
        };
    };
};
return(g);
};

/*****
Function which finds a persistent set at a
global system state with Algorithm 2 for the
cases where read-first methodology does not work
*****/

enablest* algo5(menu client, int proc, int * global)
{
    int length_of_enabled_set =0;
    enablest* rtf;
    enablest *rts;
    int OK=0;
    int l1, l2, res, h, s, red, each_state, u, hy, t;
    rel* Pp=new rel;
    rel* Pc=new rel;
    Pc->ars[0]= client.myprocess[proc].process_name;
    Pc->len++;
    l1=Pc->len;
    l2=Pp->len;
    while(!OK)
    {
        Pp=unions(Pp,Pc);
        for(h=1; h<= Pc->len; h++) //STEP 2
        {
            res=converter(client, Pc->ars[h-1]);
            if(sel == 3 )
                Pp=unions(Pp,&table[res]);
            if((sel==4 ) || (sel==0))
            {
                rel* ahyak=new rel(Pc->ars[h-1]);
                rel* f=recurse2(ahyak, global, res, client);
                Pp=unions(Pp,f);
                delete ahyak;
                delete f;
            };
        };
        for(s=1; s<=Pp->len;s++) //STEP 3
        {
            red=converter(client,Pp->ars[s-1]);
            each_state = global[red];
            for( t=1; t<=pars_transno[red];t++)
            {
                if(client.myprocess[red].mystates[t-1].state_name == each_state )

```

```

    {
        if (client.myprocess[red].mystates[t-1].inout == '-')
        {
            for( u=1; u<=Digit;u++)
            {
                if(client.myprocess[red].mystates[t-1].toprocess==
                    client.myprocess[u-1].process_name )
                    {break; };
            };
            if(client.myprocess[u-1].input_queue->mynumb < (lenq))
            {
                rel* myrel=new rel;
                myrel->len=0;
                myrel->ars[0]=client.myprocess[u-1].process_name;
                myrel->len++;
                Pc=unions(Pc, myrel);
                delete myrel;
            };
        };
    };
};

if ( (l1==Pc->len) && (l2==Pp->len) ) OK=1; //STEP 4
else {
    l1=Pc->len;
    l2=Pp->len;
};

};

for(hy=1; hy<=Pp->len;hy++)
{
    red=converter(client,Pp->ars[hy-1]);
    each_state = global[red];
    rts=enable_s(each_state, red, client);
    for( h=1; h<=rts->length_of_enabled;h++)
    {
        rtf->enabled[length_of_enabled_set] = rts->enabled[h-1] ;
        rtf->proc_set[length_of_enabled_set] = rts->proc_set[h-1];
        rtf->enabled_by_save[length_of_enabled_set] = rts->enabled_by_save[h-1];
        rtf->priority_not_enabled[length_of_enabled_set] = rts->priority_not_enabled[h-1];
        length_of_enabled_set++;
    };
};
if (sel==0)
{
    cout << Pp->ars[hy-1] << endl;
    cout << "Enabled transitions for this process is:" << endl;
    if(rts->length_of_enabled!=0)
    {
        print_to_scr(rts->enabled, rts->proc_set, rts->length_of_enabled);
    }
    else cout <<"No enabled transition for this process!" <<endl;
    cout << endl;
};
};
rts->length_of_enabled=length_of_enabled_set;
delete Pp;
delete Pc;
delete rts;
return(rtf);
};

enablest* enable_s1(int each_state, int proc, menu client)
//finds 'none' executables
{
    int len =0;
    int u, t;
    enablest* rts=new enablest;
    for( t=1; t<=pars_transno[proc];t++)

```

```

{
if(client.myprocess[proc].mystates[t-1].state_name == each_state )
{
    if (client.myprocess[proc].mystates[t-1].inout == '-')
    {
        for( u=1; u<=Digit;u++)
        {
            if(client.myprocess[proc].mystates[t-1].toprocess==
                client.myprocess[u-1].process_name )
                {break; };
        };
        if(client.myprocess[u-1].input_queue->mynumb < (lenq))
        {
            rts->enabled[len]=client.myprocess[proc].mystates[t-1];
            rts->proc_set [len]=client.myprocess [proc];
            len++;
        };
    };
    if ((client.myprocess[proc].mystates[t-1].inout == '+') &&
        ( !(client.myprocess[proc].input_queue->emptyQ())))
    {
        if(client.myprocess[proc].input_queue->headQ() ==
            client.myprocess[proc].mystates[t-1].signal)
        { rts->enabled[len]= client.myprocess[proc].mystates[t-1] ;
          rts->proc_set [len]=client.myprocess [proc];
          len++;
        };
    };
    if (client.myprocess[proc].mystates[t-1].inout == 'N')
    {
        rts->enabled[len]= client.myprocess[proc].mystates[t-1] ;
        rts->proc_set [len]=client.myprocess [proc];
        len++;
    };
};
rts->length_of_enabled = len;
delete rts;
return(rts);
};

enablest* enable1(int* present_state, menu client)
    //ONLY difference from function enable is that it
    //finds NONE transitions
{
    int length_of_enabled_set =0;
    enablest* rts=new enablest;
    enablest* retr;
    int each_state, h;
    for ( int y=1; y<=Digit; y++ )
    {
        each_state = present_state[y-1];
        retr=enable_s1(each_state, y-1, client);
        for( h=1; h<=retr->length_of_enabled;h++)
        {
            rts->enabled[length_of_enabled_set] = retr->enabled[h-1] ;
            rts->proc_set [length_of_enabled_set] = retr->proc_set [h-1];
            rts->enabled_by_save [length_of_enabled_set] = retr->enabled_by_save [h-1];
            rts->priority_not_enabled [length_of_enabled_set] = retr->priority_not_enabled [h-1];
            length_of_enabled_set++;
        };
    };
    rts->length_of_enabled=length_of_enabled_set;
    return(rts);
};

```

```

/*****
Function which finds persistent set with
read first methodology and if it cannot work
then function algo5 is called
*****/

enablest* persistent(int* present_state, menu client)
{
    enablest * rtj= new enablest;
    enablest * my=enable(present_state,client);
    int sel_pro;
    int mysel, min=1000;
    int count = 0;
    int k, u, z, r;
    if(my->length_of_enabled != 0 )
    {
        for ( r=1; r<=my->length_of_enabled; r++)
        {
            if(((my->enabled[r-1].inout == '+') &&
                ( my->priority_not_enabled[r-1] != 1)) ||
                (my->enabled[r-1].inout == 'p') )
            {
                if( sel == 0 )
                {
                    cout << "Persistent set is:" << endl;
                    cout << endl;
                    cout << "Process: " << my->proc_set[r-1].process_name << " " ;
                    cout << "State: " <<my->enabled[r-1].state_name << " " ;
                    cout << "Transition: " <<my->enabled[r-1].inout << " " ;
                    cout << my->enabled[r-1].signal << " " ;
                    cout << "To: " <<my->enabled[r-1].toprocess << " " ;
                    cout << "Next_State: " <<my->enabled[r-1].next_state_name << " " << endl;
                };
                k=0; //TAKE only one + transition
                rtj->enabled[k]= my->enabled[r-1];
                rtj->proc_set[k]=my->proc_set[r-1];
                rtj->enabled_by_save[k]=my->enabled_by_save[r-1];
                rtj->priority_not_enabled[k]=my->priority_not_enabled[r-1];
                count=count+1;
                rtj->length_of_enabled=1;
                u=converter(client, my->proc_set[r-1].process_name);
                for( z=1; z<=pars_transno[u];z++)
                {
                    if(client.myprocess[u].mystates[z-1].state_name==
                        my->enabled[r-1].state_name)
                    {
                        if ( client.myprocess[u].mystates[z-1].inout=='N')
                        {
                            k++;
                            rtj->enabled[k]=client.myprocess[u].mystates[z-1];
                            rtj->proc_set[k]=my->proc_set[r-1];
                            rtj->enabled_by_save[k]=my->enabled_by_save[r-1];
                            rtj->priority_not_enabled[k]=my->priority_not_enabled[r-1];
                            rtj->length_of_enabled++;
                        }
                    }
                };
            };
        };
    };

    if ( count == 0 ) //IF there are no + transitions
    {
        if( sel == 0 )
        {
            for(sel_pro=1; sel_pro<=Digit; sel_pro++)
            {
                rtj=algo5(client, sel_pro-1, present_state);
                cout <<"%%Tried process for persistit set is: "<<sel_pro-1<<endl;
                for ( r=1; r<=rtj->length_of_enabled; r++)

```



```

    rtj->enabled[k]=client.myprocess[u].mystates[z-1];
    rtj->proc_set[k]=my->proc_set[r-1];
    rtj->enabled_by_save[k]=my->enabled_by_save[r-1];
    rtj->priority_not_enabled[k]=my->priority_not_enabled[r-1];
    rtj->length_of_enabled++;
  }
}
};
};
};
return(rtj);
};

int hfunc1(mytype *key)
{
  int d=0;
  for(int t=1; t<=Digit;t++)
  {
    d+d*key->stateno[t-1]*expo(10,Digit-t);
  };
  return (d % (HASHSIZE - 1));
};

int statenom =0;
int isin(link** H,mytype *key)
{
  int t;
  int val=0;
  int d=0;
  link * p;
  link* q ;
  link* s;
  bool found;
  found = false;
  int i=hfunc1(key);
  q=0;
  p=H[i];
  while( (p!=0) && (!found) )
  {
    for(t=1; t<=Digit;t++)
    {
      if ( (p->a->stateno[t-1] == key->stateno[t-1] ) &&
          (length_of_queue(p->a->global_queue[t-1]) ==
           length_of_queue(key->global_queue[t-1]) ) )
        val=val+1;
    };
    if(val==Digit)
    { found =true; }
    else { val=0;
          q=p;
          p=p->next;
        };
  };
  if (found)
  {
    d=1;
    delete key;
  }
  else {
    hnum[i]++;
    if (hnum[i] > maxh)
      maxh=hnum[i];
    s=new link;
    s->a=key;
    if (q==0) H[i]=s;
    else q->next = s;
    statenom = statenom + 1;
  };
};

```

```

return d;
};

/*****
Function which uses CRA or PRA to find
the reachability tree for verification purposes and
checks whether there are deadlocks or unspecified
receptions in the system
*****/

void exhaust(int* s0, menu client)
{
    fstream stackout, hashout;
    int stsize=0;
    int transnom=0;
    int dead=0;
    int ur=0;
    int cntr=0;
    int cntr2=0;
    struct rusage time1,time2;
    int a, j, t,y,g, num, u, k, length_of_enabled_set, cou;
    H=new link*[HASHSIZE];
    enablest* my;
    state* enabled_set;
    processes* pro_set;
    mytype* nok;
    stack * mystack= newstack(STACKSIZE);
    mytype aft, *s;
    int myh=0;
    for(g=1;g<=HASHSIZE;g++)
        H[g-1]=0;
    for( u=1; u<=Digit;u++)
    {
        aft.stateno[u-1]=s0[u-1];
        aft.global_queue[u-1]=new int[length_of_queue(pars_queue[u-1])] ;
        aft.global_queue[u-1]= pars_queue[u-1];
    };
    push(mystack, &aft);
    myh++;
    getrusage(RUSAGE_SELF,&time1);
    while (!isempty(mystack))
    {
        cntr++;
        cntr2++;
        s=pop(mystack);
        myh--;
        if(!isin(H,s))
        {
            for (g=1;g<=Digit;g++)
            {
                client.myprocess[g-1].input_queue->clearQ();
                for( y=1; y<=length_of_queue(s->global_queue[g-1]); y++)
                {
                    if(client.myprocess[g-1].input_queue->mynumb < lenq )
                        client.myprocess[g-1].input_queue->enqueue(s->global_queue[g-1][y-1]);
                };
            };
        };

        if (sel==2) // CRA
            my=enable(s->stateno, client);
        if (sel==4) // PRA
            my=persistent(s->stateno, client);
        length_of_enabled_set=my->length_of_enabled;
        enabled_set=my->enabled;
        pro_set=my->proc_set;
        nok =new mytype[length_of_enabled_set];
        transnom=transnom+length_of_enabled_set;

        if (length_of_enabled_set==0)

```

```

{
  cou = 0;
  for (g=1;g<=Digit;g++)
  if (client.myprocess[g-1].input_queue->emptyQ())
    cou = cou +1;
  if (cou == Digit )
    { dead=dead+1; }
  else
    { ur=ur+1; };
};
int check ;
for( num=1; num<=length_of_enabled_set;num++)
{
  check = 0;
  for( u=1; u<=Digit;u++)
  {
    nok[num-1].stateno[u-1]= s->stateno[u-1];
    nok[num-1].global_queue[u-1]=new int[lenq];
    for(j=1;j<=lenq;j++)
    {
      nok[num-1].global_queue[u-1][j-1]=s->global_queue[u-1][j-1];
    };
  };
  for( k=1; k<=Digit;k++)
  {
    if(client.myprocess[k-1].process_name ==
      pro_set[num-1].process_name )
    { break; };
  };
  if (enabled_set[num-1].inout == '-')
  {
    nok[num-1].stateno[k-1]=enabled_set[num-1].next_state_name;
    for(u=1; u<=Digit;u++)
    {
      if(client.myprocess[u-1].process_name==
        enabled_set[num-1].toprocess )
      {
        a = length_of_queue(s->global_queue[u-1]);
        nok[num-1].global_queue[u-1][a]=enabled_set[num-1].signal;
      };
    };
  }
  else if((enabled_set[num-1].inout == '+') &&
    ( my->enabled_by_save[num-1] == 0 )
    )
  {
    nok[num-1].stateno[k-1]=enabled_set[num-1].next_state_name;
    a = length_of_queue(s->global_queue[k-1]);
    for(j=1;j<=a;j++)
    {
      nok[num-1].global_queue[k-1][j-1]=s->global_queue[k-1][j];
    };
    nok[num-1].global_queue[k-1][a-1]=0;
  }
  else if((enabled_set[num-1].inout == '+') &&
    ( my->enabled_by_save[num-1] == 1 )
    )
  {
    nok[num-1].stateno[k-1]=enabled_set[num-1].next_state_name;
    a = length_of_queue(s->global_queue[k-1]);
    for(j=1;j<=a;j++)
    {
      if((isinSave(s->global_queue[k-1][j-1],
        enabled_set[num-1].save) ) && ( check == 0 ) )
        nok[num-1].global_queue[k-1][j-1]=s->global_queue[k-1][j-1];
      else
      {
        nok[num-1].global_queue[k-1][j-1]=s->global_queue[k-1][j];
        check=1;
      }
    }
  }
}

```

```

    };
    nok[num-1].global_queue[k-1][a-1]=0;
    };
    }
else if(enabled_set[num-1].inout == 'p')
{
    nok[num-1].stateno[k-1]=enabled_set[num-1].next_state_name;
    a = length_of_queue(s->global_queue[k-1]);
    for(j=1;j<=a;j++)
    {
        if(!(s->global_queue[k-1][j-1] ==
            enabled_set[num-1].signal)
            && ( check == 0 ) )
            nok[num-1].global_queue[k-1][j-1]=s->global_queue[k-1][j-1];
        else
        {
            nok[num-1].global_queue[k-1][j-1]=s->global_queue[k-1][j];
            check=1;
        };
        nok[num-1].global_queue[k-1][a-1]=0;
    };
    };
else if(enabled_set[num-1].inout == 'N')
{
    nok[num-1].stateno[k-1]=enabled_set[num-1].next_state_name;
    };

if (length_of_enabled_set!=0)
{
    push(mystack,&(nok[num-1]));
    myh++;
    stsize = stsize+1;
    };
};
delete [] enabled_set;
delete [] pro_set;
delete my;
};

if (cntr==10000)
{
    system("pstat -s");
    printf("\rStack size: %d Trans num: %d
        States: %d DEADLOCK: %d UR: %d",myh,stsize,statenom,dead,ur);
    cntr=0;
    cout << endl;
    cout << en << endl;
    cout << pr << endl;
    cout << lnk << endl;
    cout << stn << endl;
    cout << gq << endl;
    cout << relno << endl;
    };
    };
getrusage(RUSAGE_SELF,&time2);
cout << "RESULTS" << endl;
cout << "-----" << endl;
cout << "Blocking state number: " << dead+ur << endl;
cout << "Deadlock number: " << dead << endl;
cout << "UR number: " << ur << endl;
cout << "Global state number: " << statenom << endl;
cout << "Global transition number: " << transnom << endl;
cout << "User Time: " <<
    time2.ru_utime.tv_sec - time1.ru_utime.tv_sec << endl;
cout << "System Time: " <<
    time2.ru_stime.tv_sec - time1.ru_stime.tv_sec << endl;
cout << "Maximum hash link = " << maxh << endl;
cout << en << endl;
cout << pr << endl;

```

```

cout << lnk << endl;
cout << stn << endl;
cout << gq << endl;
cout << relno << endl;
};

/*****
Main function which makes necessary initializations
like asking the user to enter the name of the protocol
to be verified and assigning a bound on the input queues
of processes and running function exhaustfor CRA or PRA
*****/

void main(){
top2:
int hh=0;
maxh=0;
for(hh=0;hh<HASHSIZE;hh++)
    hnum[hh]=0;
myfile= initialize1();
cout << "Enter length of queue: " ;
cin >> lenq ;
menu client;
time_t first, second;
table=new rel[Digit];
tablep=new rel[Digit];
relationstar(client);
for(int in=1; in<=Digit; in++)
{
cout << "Table" << endl;
//This table illustrates the graph G_R in the thesis
for(int jk=1; jk<=table[in-1].len; jk++)
{
cout << table[in-1].ars[jk-1];
};
cout << endl;
};
top1:
int* global=initialize2();
switch(sel)
{
case 2:
case 4: { first = time(NULL);
exhaust(global, client);
second = time(NULL);
cout << "Time passed: " << (second- first) << endl;
break; };
case 5: { exit(1); break;};
};
int choice;
cout << "Enter a new file or continue executing a trans (0/1)" ;
cin >> choice;
if(choice)
{
goto top1;
}
else goto top2;
}

```

# APPENDIX D

## QUEUE\_INT.H SOFTWARE CODE

```
#include      <stdlib.h>

class int_queue
{
    int* queue;
public:
    int mynumb;
    int_queue(int nummembers);
    int_queue();
    ~int_queue();
    void enqueue(int s);
    int dequeue();
    int emptyQ(){return (mynumb == 0); }
    void clearQ();
    int headQ();
    int isinQ(int s);
    int saveQ(int s, int* savelist);
};

int int_queue:: isinQ(int s)
{
    int ret=0;
    for(int g=1; g<=mynumb; g++)
    {
        if ( queue[g-1] == s )
            { ret=g ; break;};
    }
    return(ret);
}

/* checks whether s is after saved signals */

int int_queue:: saveQ(int s, int* savelist)
{
    int ret=1;
    int k=isinQ(s);
    if ( ( k>1 )  && ( !isinSave(s, savelist) ) )
    {
        for(int g=1; g<=k-1; g++)
        {
            if ( !isinSave(queue[g-1], savelist) ) ret=0;
        };
    }
    else ret=0;
}
```

```

    return(ret);
}

int_queue:: int_queue(int nummembers)
{
    queue=new int[nummembers];
    mynumb=0;
}

int_queue::~ ~int_queue()
{
    delete queue;
}

void int_queue:: enqueue(int c)
{
    queue[mynumb] = c;
    mynumb = mynumb + 1;
    if (mynumb == nummembers)
        cout << " Overflow of character queue " << endl;
}

int int_queue:: dequeue()
{
    if(emptyQ())
        cout << " Underflow of character queue" << endl;

    int ch = queue[0];
    mynumb=mynumb-1;
    for(int h=1;h<=mynumb;h++)
    {
        queue[h-1]=queue[h];
    }
    return ch;
}

void int_queue:: clearQ()
{
    mynumb=0;
}

int int_queue:: headQ()
{
    int ch = queue[0];
    return ch;
}

```

# APPENDIX E

## STACK.H SOFTWARE CODE

```
struct mytype{
int* stateno ;
int** global_queue;
mytype(){
    global_queue=new int*[Digit];
    stateno=new int[Digit];
    gq++; stn++;
};
~mytype(){
    delete [] global_queue;
    delete [] stateno;
    gq--; stn--;
};
};

typedef struct
{
    int maxsize;
    int curtop;
    mytype **mem;
} stack;

void clearstack(stack *s)
{
    s->curtop=0;
};

int isempty(stack *s)
{
    return s->curtop == 0;
};

stack * newstack(int maxsize)
{
    stack *s=new stack;
    s->maxsize=maxsize;
    s->mem=new mytype*[maxsize];
    clearstack(s);
    return s;
};

void delstack(stack *s)
{
    delete s->mem;
```

```

    delete s;
};

void push(stack *s, mytype *c)
{
    if (s->curtop < s->maxsize)
    {
        s->mem[s->curtop]=c;
        s->curtop++;
    }
    else
        cout << "stack overflow" << endl;
};

mytype * pop(stack *s)
{
    if (s->curtop > 0 )
        return s->mem[--s->curtop];
    else
        cout << "stack underflow" << endl;
};

mytype * top(stack *s)
{
    if (s->curtop > 0 )
        return s->mem[s->curtop-1];
    else
        cout << "stack underflow" << endl;
};

```

# APPENDIX F

## ISDN SPECIFIED IN INPUT LANGUAGE OF POVSDL

```
Process_number 6
Process 3
Input_queue
Transition_number 250
State 0
Save
+ 1
Next_State 1
State 0
Save
+ 2
Next_State 0
State 0
Save
+ 3
Next_State 0
State 0
Save
+ 4
Next_State 12
State 0
Save
+ 13
Next_State 3
State 0
Save
+ 14
Next_State 0
State 0
Save
+ 15
Next_State 0
State 0
Save
+ 33
Next_State 0
State 0
Save
+ 36
Next_State 0
State 0
Save
+ 38
Next_State 0
```

```
State 0
Save
+ 39
Next_State 0
State 0
Save
+ 40
Next_State 0
State 0
Save
+ 41
Next_State 0
State 0
Save
+ 42
Next_State 0
State 0
Save
+ 43
Next_State 0
State 0
Save
+ 44
Next_State 0
State 0
Save
+ 45
Next_State 0
State 0
Save
+ 46
Next_State 0
State 0
Save
+ 47
Next_State 0
State 0
Save
+ 66
Next_State 0
State 0
Save
+ 67
Next_State 0
State 0
Save
+ 75
Next_State 0
State 0
Save
+ 76
Next_State 0
State 12
Save
- 67 toProcess 3
Next_State 2
State 1
Save 66$67$
+ 1
Next_State 1
State 1
Save 66$67$
+ 2
Next_State 1
State 1
Save 66$67$
+ 3
Next_State 1
State 1
```

Save 66\$67\$  
+ 4  
Next\_State 13  
State 1  
Save 66\$67\$  
+ 13  
Next\_State 205  
State 1  
Save 66\$67\$  
+ 14  
Next\_State 0  
State 1  
Save 66\$67\$  
+ 15  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 33  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 36  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 38  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 39  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 40  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 41  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 42  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 43  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 44  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 45  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 46  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 47  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 66  
Next\_State 1  
State 1  
Save 66\$67\$

+ 67  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 75  
Next\_State 1  
State 1  
Save 66\$67\$  
+ 76  
Next\_State 1  
State 13  
Save  
- 67 toProcess 3  
Next\_State 1  
State 205  
Save  
- 68 toProcess 6  
Next\_State 206  
State 206  
Save  
- 71 toProcess 7  
Next\_State 4  
State 2  
Save 67\$  
+ 1  
Next\_State 1  
State 2  
Save 67\$  
+ 2  
Next\_State 2  
State 2  
Save 67\$  
+ 3  
Next\_State 2  
State 2  
Save 67\$  
+ 4  
Next\_State 12  
State 2  
Save 67\$  
+ 13  
Next\_State 3  
State 2  
Save 67\$  
+ 14  
Next\_State 0  
State 2  
Save 67\$  
+ 15  
Next\_State 0  
State 2  
Save 67\$  
+ 33  
Next\_State 2  
State 2  
Save 67\$  
+ 36  
Next\_State 2  
State 2  
Save 67\$  
+ 38  
Next\_State 2  
State 2  
Save 67\$  
+ 39  
Next\_State 2  
State 2  
Save 67\$  
+ 40

Next\_State 2  
State 2  
Save 67\$  
+ 41  
Next\_State 2  
State 2  
Save 67\$  
+ 42  
Next\_State 2  
State 2  
Save 67\$  
+ 43  
Next\_State 2  
State 2  
Save 67\$  
+ 44  
Next\_State 2  
State 2  
Save 67\$  
+ 45  
Next\_State 2  
State 2  
Save 67\$  
+ 46  
Next\_State 2  
State 2  
Save 67\$  
+ 47  
Next\_State 2  
State 2  
Save 67\$  
+ 66  
Next\_State 2  
State 2  
Save 67\$  
+ 67  
Next\_State 2  
State 2  
Save 67\$  
+ 75  
Next\_State 2  
State 2  
Save 67\$  
+ 76  
Next\_State 2  
State 3  
Save  
+ 1  
Next\_State 205  
State 3  
Save  
+ 2  
Next\_State 3  
State 3  
Save  
+ 3  
Next\_State 3  
State 3  
Save  
+ 4  
Next\_State 20  
State 3  
Save  
+ 13  
Next\_State 3  
State 3  
Save  
+ 14  
Next\_State 3

State 3  
Save  
+ 15  
Next\_State 0  
State 3  
Save  
+ 33  
Next\_State 3  
State 3  
Save  
+ 36  
Next\_State 92  
State 3  
Save  
+ 38  
Next\_State 18  
State 3  
Save  
+ 39  
Next\_State 90  
State 3  
Save  
+ 40  
Next\_State 3  
State 3  
Save  
+ 41  
Next\_State 3  
State 3  
Save  
+ 42  
Next\_State 3  
State 3  
Save  
+ 43  
Next\_State 3  
State 3  
Save  
+ 44  
Next\_State 3  
State 3  
Save  
+ 45  
Next\_State 3  
State 3  
Save  
+ 46  
Next\_State 3  
State 3  
Save  
+ 47  
Next\_State 3  
State 3  
Save  
+ 66  
Next\_State 3  
State 3  
Save  
+ 67  
Next\_State 3  
State 3  
Save  
+ 75  
Next\_State 3  
State 3  
Save  
+ 76  
Next\_State 3  
State 90

Save  
NONE  
Next\_State 3  
State 90  
Save  
NONE  
Next\_State 205  
State 92  
Save  
NONE  
Next\_State 243  
State 92  
Save  
NONE  
Next\_State 3  
State 18  
Save  
NONE  
Next\_State 278  
State 18  
Save  
NONE  
Next\_State 3  
State 278  
Save  
- 19 toProcess 4  
Next\_State 3  
State 20  
Save  
- 67 toProcess 3  
Next\_State 3  
State 4  
Save 2\$  
+ 1  
Next\_State 4  
State 4  
Save 2\$  
+ 2  
Next\_State 4  
State 4  
Save 2\$  
+ 3  
Next\_State 120  
State 4  
Save 2\$  
+ 4  
Next\_State 31  
State 4  
Save 2\$  
+ 13  
Next\_State 4  
State 4  
Save 2\$  
+ 14  
Next\_State 4  
State 4  
Save 2\$  
+ 15  
Next\_State 208  
State 4  
Save 2\$  
+ 75  
Next\_State 109  
State 4  
Save 2\$  
+ 33  
Next\_State 4  
State 4  
Save 2\$

+ 36  
Next\_State 4  
State 4  
Save 2\$  
+ 38  
Next\_State 100  
State 4  
Save 2\$  
+ 39  
Next\_State 107  
State 4  
Save 2\$  
+ 40  
Next\_State 4  
State 4  
Save 2\$  
+ 41  
Next\_State 4  
State 4  
Save 2\$  
+ 42  
Next\_State 4  
State 4  
Save 2\$  
+ 43  
Next\_State 4  
State 4  
Save 2\$  
+ 44  
Next\_State 4  
State 4  
Save 2\$  
+ 45  
Next\_State 4  
State 4  
Save 2\$  
+ 66  
Next\_State 4  
State 4  
Save 2\$  
+ 67  
Next\_State 4  
State 4  
Save 2\$  
+ 76  
Next\_State 4  
State 208  
Save  
- 69 toProcess 6  
Next\_State 0  
State 100  
Save  
NONE  
Next\_State 4  
State 100  
Save  
NONE  
Next\_State 209  
State 209  
Save  
- 69 toProcess 6  
Next\_State 210  
State 210  
Save  
- 70 toProcess 7  
Next\_State 6  
State 107  
Save  
NONE

Next\_State 4  
State 107  
Save  
NONE  
Next\_State 108  
State 108  
Save  
- 69 toProcess 6  
Next\_State 3  
State 109  
Save  
NONE  
Next\_State 256  
State 109  
Save  
NONE  
Next\_State 111  
State 256  
Save  
- 68 toProcess 6  
Next\_State 4  
State 111  
Save  
- 23 toProcess 4  
Next\_State 3  
State 120  
Save  
NONE  
Next\_State 4  
State 120  
Save  
NONE  
Next\_State 280  
State 280  
Save  
- 66 toProcess 3  
Next\_State 4  
State 31  
Save  
- 67 toProcess 3  
Next\_State 4  
State 5  
Save  
+ 1  
Next\_State 5  
State 5  
Save  
+ 2  
Next\_State 5  
State 5  
Save  
+ 3  
Next\_State 5  
State 5  
Save  
+ 4  
Next\_State 42  
State 5  
Save  
+ 13  
Next\_State 5  
State 5  
Save  
+ 14  
Next\_State 5  
State 5  
Save  
+ 15  
Next\_State 212

State 5  
Save  
+ 33  
Next\_State 5  
State 5  
Save  
+ 36  
Next\_State 5  
State 5  
Save  
+ 38  
Next\_State 130  
State 5  
Save  
+ 39  
Next\_State 131  
State 5  
Save  
+ 40  
Next\_State 5  
State 5  
Save  
+ 41  
Next\_State 5  
State 5  
Save  
+ 42  
Next\_State 5  
State 5  
Save  
+ 43  
Next\_State 5  
State 5  
Save  
+ 44  
Next\_State 5  
State 5  
Save  
+ 45  
Next\_State 5  
State 5  
Save  
+ 46  
Next\_State 5  
State 5  
Save  
+ 47  
Next\_State 5  
State 5  
Save  
+ 66  
Next\_State 5  
State 5  
Save  
+ 67  
Next\_State 5  
State 5  
Save  
+ 75  
Next\_State 135  
State 5  
Save  
+ 76  
Next\_State 5  
State 212  
Save  
- 69 toProcess 6  
Next\_State 0  
State 213

Save  
- 69 toProcess 6  
Next\_State 3  
State 130  
Save  
- 69 toProcess 6  
Next\_State 3  
State 131  
Save  
NONE  
Next\_State 213  
State 131  
Save  
NONE  
Next\_State 5  
State 135  
Save  
- 24 toProcess 4  
Next\_State 3  
State 214  
Save  
- 68 toProcess 6  
Next\_State 5  
State 42  
Save  
- 67 toProcess 3  
Next\_State 5  
State 215  
Save  
- 71 toProcess 7  
Next\_State 214  
State 49  
Save  
- 66 toProcess 3  
Next\_State 6  
State 185  
Save  
NONE  
Next\_State 281  
State 185  
Save  
NONE  
Next\_State 6  
State 281  
Save  
- 71 toProcess 7  
Next\_State 217  
State 217  
Save  
- 68 toProcess 6  
Next\_State 6  
State 218  
Save  
- 68 toProcess 6  
Next\_State 219  
State 219  
Save  
- 71 toProcess 7  
Next\_State 7  
State 6  
Save  
+ 1  
Next\_State 205  
State 6  
Save  
+ 2  
Next\_State 215  
State 6  
Save

+ 3  
Next\_State 49  
State 6  
Save  
+ 4  
Next\_State 59  
State 6  
Save  
+ 13  
Next\_State 6  
State 6  
Save  
+ 14  
Next\_State 6  
State 6  
Save  
+ 15  
Next\_State 224  
State 6  
Save  
+ 33  
Next\_State 205  
State 6  
Save  
+ 36  
Next\_State 219  
State 6  
Save  
+ 38  
Next\_State 54  
State 6  
Save  
+ 39  
Next\_State 147  
State 6  
Save  
+ 40  
Next\_State 205  
State 6  
Save  
+ 41  
Next\_State 151  
State 6  
Save  
+ 42  
Next\_State 205  
State 6  
Save  
+ 43  
Next\_State 205  
State 6  
Save  
+ 44  
Next\_State 205  
State 6  
Save  
+ 45  
Next\_State 205  
State 6  
Save  
+ 46  
Next\_State 205  
State 6  
Save  
+ 47  
Next\_State 205  
State 6  
Save  
+ 66

Next\_State 185  
State 6  
Save  
+ 67  
Next\_State 6  
State 6  
Save  
+ 75  
Next\_State 218  
State 6  
Save  
+ 76  
Next\_State 218  
State 219  
Save  
- 69 toProcess 6  
Next\_State 220  
State 220  
Save  
- 69 toProcess 6  
Next\_State 221  
State 221  
Save  
- 71 toProcess 7  
Next\_State 3  
State 54  
Save  
- 19 toProcess 4  
Next\_State 6  
State 147  
Save  
NONE  
Next\_State 205  
State 147  
Save  
NONE  
Next\_State 6  
State 151  
Save  
NONE  
Next\_State 209  
State 151  
Save  
NONE  
Next\_State 153  
State 151  
Save  
NONE  
Next\_State 205  
State 153  
Save  
- 68 toProcess 6  
Next\_State 6  
State 224  
Save  
- 69 toProcess 6  
Next\_State 225  
State 225  
Save  
- 71 toProcess 7  
Next\_State 0  
State 59  
Save  
- 67 toProcess 3  
Next\_State 6  
State 7  
Save 66\$  
+ 1  
Next\_State 205

State 7  
Save 66\$  
+ 2  
Next\_State 234  
State 7  
Save 66\$  
+ 3  
Next\_State 67  
State 7  
Save 66\$  
+ 4  
Next\_State 77  
State 7  
Save 66\$  
+ 13  
Next\_State 7  
State 7  
Save 66\$  
+ 14  
Next\_State 7  
State 7  
Save 66\$  
+ 15  
Next\_State 238  
State 7  
Save 66\$  
+ 33  
Next\_State 205  
State 7  
Save 66\$  
+ 36  
Next\_State 242  
State 7  
Save 66\$  
+ 38  
Next\_State 68  
State 7  
Save 66\$  
+ 39  
Next\_State 205  
State 7  
Save 66\$  
+ 40  
Next\_State 205  
State 7  
Save 66\$  
+ 41  
Next\_State 181  
State 7  
Save 66\$  
+ 42  
Next\_State 181  
State 7  
Save 66\$  
+ 43  
Next\_State 181  
State 7  
Save 66\$  
+ 44  
Next\_State 205  
State 7  
Save 66\$  
+ 45  
Next\_State 205  
State 7  
Save 66\$  
+ 46  
Next\_State 205  
State 7

```
Save 66$
+ 47
Next_State 205
State 7
Save 66$
+ 66
Next_State 7
State 7
Save 66$
+ 67
Next_State 7
State 7
Save 66$
+ 75
Next_State 205
State 7
Save 66$
+ 76
Next_State 7
State 234
Save
- 68 toProcess 6
Next_State 235
State 235
Save
- 71 toProcess 7
Next_State 5
State 67
Save
- 66 toProcess 3
Next_State 7
State 68
Save
- 19 toProcess 4
Next_State 7
State 238
Save
- 69 toProcess 6
Next_State 239
State 239
Save
- 71 toProcess 7
Next_State 0
State 242
Save
- 69 toProcess 6
Next_State 243
State 243
Save
- 70 toProcess 7
Next_State 6
State 181
Save
- 69 toProcess 6
Next_State 248
State 248
Save
- 70 toProcess 7
Next_State 202
State 77
Save
- 67 toProcess 3
Next_State 7
End_process 3

Process 4
Input_queue
Transition_number 24
State 0
```

Save  
+ 19  
Next\_State 0  
State 0  
Save  
+ 23  
Next\_State 0  
State 0  
Save  
+ 24  
Next\_State 0  
State 0  
Save  
+ 60  
Next\_State 0  
State 0  
Save  
+ 61  
Next\_State 0  
State 0  
Save  
+ 62  
Next\_State 3  
State 0  
Save  
+ 64  
Next\_State 0  
State 0  
Save  
+ 65  
Next\_State 0  
State 0  
Save  
+ 74  
Next\_State 4  
State 1  
Save  
+ 19  
Next\_State 7  
State 1  
Save  
+ 23  
Next\_State 7  
State 1  
Save  
+ 24  
Next\_State 7  
State 1  
Save  
+ 60  
Next\_State 3  
State 1  
Save  
+ 61  
Next\_State 7  
State 1  
Save  
+ 62  
Next\_State 1  
State 1  
Save  
+ 64  
Next\_State 7  
State 1  
Save  
+ 65  
Next\_State 7  
State 1  
Save

```
+ 74
Next_State 7
State 3
Save
- 13 toProcess 3
Next_State 259
State 259
Save
- 73 toProcess 5
Next_State 1
State 4
Save
- 72 toProcess 5
Next_State 290
State 290
Save
- 14 toProcess 3
Next_State 0
State 7
Save
- 73 toProcess 5
Next_State 263
State 263
Save
- 15 toProcess 3
Next_State 0
End_process 4

Process 5
Input_queue
Transition_number 6
State 0
Save
+ 72
Next_State 1
State 0
Save
+ 73
Next_State 0
State 1
Save
+ 73
Next_State 0
State 1
Save
+ 72
Next_State 1
State 1
Save
NONE
Next_State 2
State 2
Save
- 74 toProcess 4
Next_State 0
End_process 5

Process 6
Input_queue
Transition_number 6
State 0
Save
+ 68
Next_State 1
State 0
Save
+ 69
Next_State 0
State 1
```

```

Save
+ 69
Next_State 0
State 1
Save
+ 68
Next_State 1
State 1
Save
NONE
Next_State 2
State 2
Save
- 75 toProcess 3
Next_State 0
End_process 6

Process 7
Input_queue
Transition_number 6
State 0
Save
+ 70
Next_State 1
State 0
Save
+ 71
Next_State 0
State 1
Save
+ 71
Next_State 0
State 1
Save
+ 70
Next_State 1
State 1
Save
NONE
Next_State 2
State 2
Save
- 76 toProcess 3
Next_State 0
End_process 7

Process 9
Input_queue
Transition_number 21
State 0
Save
- 60 toProcess 4
Next_State 1
State 1
Save
- 62 toProcess 4
Next_State 3
State 3
Save
- 64 toProcess 4
Next_State 4
State 4
Save
- 65 toProcess 4
Next_State 5
State 5
Save
- 61 toProcess 4
Next_State 6

```

State 6  
Save  
- 1 toProcess 3  
Next\_State 7  
State 7  
Save  
- 2 toProcess 3  
Next\_State 8  
State 8  
Save  
- 3 toProcess 3  
Next\_State 9  
State 9  
Save  
- 4 toProcess 3  
Next\_State 12  
State 12  
Save  
- 33 toProcess 3  
Next\_State 14  
State 14  
Save  
- 36 toProcess 3  
Next\_State 16  
State 16  
Save  
- 38 toProcess 3  
Next\_State 17  
State 17  
Save  
- 39 toProcess 3  
Next\_State 18  
State 18  
Save  
- 40 toProcess 3  
Next\_State 19  
State 19  
Save  
- 41 toProcess 3  
Next\_State 20  
State 20  
Save  
- 42 toProcess 3  
Next\_State 21  
State 21  
Save  
- 43 toProcess 3  
Next\_State 22  
State 22  
Save  
- 44 toProcess 3  
Next\_State 23  
State 23  
Save  
- 45 toProcess 3  
Next\_State 24  
State 24  
Save  
- 46 toProcess 3  
Next\_State 25  
State 25  
Save  
- 47 toProcess 3  
Next\_State 25  
End\_process 9