Université de Montréal

State Abstraction in SDL

par

Hu Yun

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté a la Faculté des études supérieures

En vue de l'obtention du grade de

Maître en Informatique

June, 2001

Université de Montréal

QA
76
J54
2001
v.022

Faculté des études supérieures

Ce mémoire de maîtrise intitulé

State Abstraction in SDL

Présenté par

Hu Yun

A été évalué par un jury composé des personnes suivantes:

Présidente:	Sahraoui Houari
Directeur de recherche:	Rudolf K. Keller
Co-Directeur de recherche:	Alexandre Petrenko
Membre:	Valtchev Petko

Mémoire accepté le: 24 juillet 2001

Acknowledgments

I would like to express my sincere appreciation to my supervisor, Professor Alex Petrenko for his invaluable guidance, his encouragement, his care and support throughout the course of this thesis work. I wish to thank my thesis director in Université de Montréal, Professor Rudolf K. Keller. The thesis is benefited from their careful reading and constructive criticism.

I would like to have a special thank to Dr. Serge Boroday for his collaboration in the development and implementation of the Algorithms. I benifited greatly from formal and informal discussion with him.

I also truly thank CRIM (Centre de recherche informatique de Montreal). It provides wonderful research environment and financial support for this research.

I wish to thank the Département d'Informatique et recherche opérationnelle, Université de Montréal for the graduate courses and research environment. Thanks to Mariette Paradis for easing the procedure of dealing with the Département.

I also express my thanks to my friends Ye Haiwei, Guo Yong, Xu Ying, Zhang Yuan, Qiu Jun, Lu Rong and Gao Liqian for their help, friendship and encouragement.

Finally, I wish to express my special acknowledgment to my husband Luo Qing and my son for their support, understanding, and making all of this possible.

Résumé

Le langage LDS (in English SDL) est largement utilisé pour modéliser des systèmes temps-réel. Pourtant, simuler des gros systèmes de LDS prend généralement un temps exponentiel par rapport au nombre des états dans la spécification. Afin de réduire le temps de simulation, certains états peuvent être fusionnés (abstracted). Le système résultant contient moins d'états et peut être considéré comme une abstraction du système original. Comme toute abstraction, une telle simplification des spécifications de LDS peut mener à des spécifications moins précises que les originales. Cependant, si cela facilite le traitement du système avec les outils existants, les efforts se trouvent justifiés.

Dans ce mémoire, nous étudions des techniques pragmatiques pour l'abstraction des états et pour l'observation des états EFSM/LDS et nous cherchons une façon possible d'abstraire les spécifications de LDS en fusionnant les états. Pour valider les techniques proposées nous implantons un ensemble d'outils expérimentaux pour l'abstraction et l'observation des états à l'aide du langage fonctionnel Caml. À partir d'une spécification réelle de LDS, nous démontrons que les outils développés réussissent avec des spécifications réelles de LDS et permettent souvent de simplifier la spécification originale.

Abstract

The SDL language is widely used to model real time systems. To simulate real size SDL systems it usually takes time exponential to the number of states in the specification. In order to reduce simulation time, some SDL states may be merged, i.e., abstracted. The resulting system with fewer states can be considered as an abstraction of the given system. As any other abstractions, such a simplification of SDL specifications could result in specifications that are less accurate than the original one. If, however, the system becomes more tractable with existing simulation tools, then these efforts are well justified.

In this thesis, we consider pragmatic techniques for state abstraction and for EFSM/SDL state observabilization and investigate one possible way of abstracting SDL specifications by state merging. To validate the proposed techniques we implement a set of experimental tools for state abstraction and state observabilization using the functional language Caml. Using a real SDL specification, we demonstrate that the developed tools perform well on a real SDL specification and often allow one to simplify the original specification.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The specification and description language (SDL) is an object-oriented, formal language defined by the International Telecommunications Union–Telecommunications Standardization Sector (ITU–T) as recommendation Z.100 [CCITT92]. The language SDL is used to produce formal specifications of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals. It is especially well suited for specification of reactive systems in general and communications protocols, in particular. One of the main advantages of using the SDL is that an SDL specification can be validated using a tool by exhaustively simulating its possible executions to reveal any design flaw. At the same time, simulating these systems is not an easy task even for an advanced simulation environment or model checker.

Real size SDL systems usually require simulation time exponential to the number of states in the specification. To alleviate the state explosion effect, one could perform certain abstractions of the original SDL specification by removing some specification elements (and possibly replacing them by simpler ones).

Abstraction methodology is rapidly developing in soft/hardware design, modeling, verification, and testing. Typically, an abstraction is intended to simplify the original specification, while preserving a number of essential properties. An abstraction could result in specifications that are less accurate than the original one. If, however, the system becomes more tractable with the existing simulation tools, the efforts pay off.

While general principles underlying abstraction are known, not much work has been done to apply them to do abstractions in systems specified in SDL. Abstraction of an SDL process can be accomplished in several ways, however, in this work we limit ourselves to so-called state abstraction. In order to define abstraction techniques appropriate for the SDL language, we need to elaborate methods for state abstraction in terms of finite state machine (FSM) and extended finite state machine (EFSM), since they form a basis for this language. Moreover, as we are often interested in observable external behavior (or set of input output sequences) of an EFSM, i.e., an SDL process, we pay special attention to abstraction which preserves behavior of the original system. Another important aspect of abstraction is a so-called observability problem. A (nondeterministic) state machine is said to be non observable if after some input output sequence of external events it may end up in several different states. This property is usually undesirable in systems, so one has to make certain transformation to eliminate it; this is what is called by observabilization techniques. Performing state abstraction, we have to observabilize the transformed system.

Thus, the purpose of this work is to

- Develop techniques for merging states in the FSM and EFSM models.
- Adapt the above techniques to a single SDL process to construct a simplified SDL specification.
- Develop basic techniques for observabilizing the SDL specifications.

- Implement the techniques for SDL state abstraction in a tool within the ObjectGeode environment.
- Conduct a case study to demonstrate the applicability of the developed tool and to illustrate the effects of SDL state abstraction.

The thesis is organized in seven chapters as follows.

Chapter 2 gives a review of the related theory of formal methods and the concepts of the finite state machine and extended finite state machine. It also introduces SDL language and the ObjectGeode environment, which is the tool for simulating SDL specifications.

In Chapter 3, we study state merging with respect to the preservation of the observability and behavior of the initial machine. Main state abstraction techniques are defined in terms of FSM and EFSM models.

In Chapter 4, we extend the methods developed in Chapter 3 for EFSM state abstraction and observabilization to SDL processes. The correspondence between EFSM and SDL processes is discussed first. Then the techniques of state merging and weak observabilization for the SDL process are described and the definition for SDL-machine is given. Finally it proposes the steps for weak observabilization of an SDL-machine.

In Chapter 5, we first explain our choice of Caml (Categorical Abstract Machine Language) as the implementation language and present the distinctive features of Caml. We describe the algorithms used for state abstraction, the algorithm of transformation of SDL transitions into a tree form, and the algorithm of weak observabilization. The implementation of the algorithms is based on the CAML-SDL API developed by France Telecom R&D. This API allows one to parse a textual SDL specification and extract an abstract syntax tree (AST) and pretty prints a given abstract syntax tree into a textual (pr) SDL specification. Finally, we present the details of the tool set implementation.

In Chapter 6, we demonstrate how the developed tool set performs when applied to a relatively complex example SDL specification. We also use this example in order to evaluate (in an experimental way) how state abstraction techniques developed and implemented in this thesis affect the complexity of SDL specifications. Several conclusions about the performance of our tool set and abstraction techniques are drawn from this experiment.

Finally, in Chapter 7, we summarize the thesis and propose potential future work.

Chapter 2

Finite State Machines and SDL Language

2.1 Formal Methods in System Development

In the current software development practice, systems are usually specified in a natural language such that these systems inevitably contain errors and ambiguities. Some of these flaws may not be easy to be uncovered during the verification and implementation phases. The primary goal of formal methods is to help system builders to understand what they are doing, both in terms of specification and implementation. The Encyclopedia of Software Engineering [Marc1994] defines formal methods in the following manner.

Formal methods used in developing computer systems are mathematically based techniques for describing system properties. Such formal methods provide frameworks which people can specify, develop, and verify systems in a systematic, rather than in ad hoc manner. A method is formal if it has a sound mathematical basis, typically given by a formal specification language. This basis provides a means of precisely defining notions like consistency and completeness, and, more relevant, specification, implementation, and correctness.

Formal methods are based on Formal Description Techniques (FDTs), which allow one to produce various descriptions from abstract to implementation-oriented. In the context of communication protocols, FDTs have been developed for the description of communication protocols and services. All FDTs can offer the means for producing unambiguous descriptions of systems in a more precise and comprehensive way than natural language descriptions. They provide a foundation for analysis and verification of a design. The target of analysis and verification may vary from abstract properties to concrete properties.

2.1.1 Formal Specifications

A *formal specification* usually describes the intended behavior of a system. An implementation actually implements the behavior described in the specification. In the specification we want to capture how the system should behave without being concerned with how that behavior is to be achieved. Implementing the system we are concerned with how to achieve the behavior on a particular hardware. Obviously a specification serves as a basis for deriving implementations, and it should abstract from implementation details in order to give an overview of a complex system, to postpone implementation decisions, and not to exclude valid implementations. To ensure that the implementation of the system is acceptable, it must be tested for conformance to its specification. The later errors are discovered in the process of specification, design and implementation, the more expensive the system will be.

A formal specification uses mathematical methods to model a system. It is possible to formally reason about the system, i.e., it is possible to verify that in a particular state, with particular inputs the specified system will behave in some desired way, or will not behave in some undesired way. It is essential to have a machine-readable form for the formal specification of systems. Once a machine-readable specification is available, tools can be applied to validate the system, generate the implementation code and conformance tests directly from the specification at a lower cost and in shorter time compared to a traditional system development approach [Monk1999].

FDTs, such as Estelle, LOTOS and SDL are well-known specification languages standardized by ISO [IS9074, IS8807] and ITU-T (formerly CCITT) [CCITT92]. They are based on the finite state machine models. The use of specification languages makes it possible to analyze and simulate system solutions, which in practice is impossible when using a programming language due to the cost and the time delay.

The requirement that a formal specification be expressed abstractly usually means that a formal specification language can not be executed directly. That is, in contrast to a program, a formal specification (that is a specification written in a specification language such as SDL) is not intended to be run on a computer. The main benefit of using a formal specification is to gain a deeper understanding of the system being specified. It is through the specification and validation process that the developers uncover design flaws, inconsistencies, ambiguities, and incompleteness [CW1996].

2.1.2 Verification and Validation

The use of formal specifications provides the basis for allowing *validation* of the specification towards expected behavior and *verification* of an implementation according to the formal specification. The testing of an implementation is different from the testing of a formal specification. Implementation testing is also called verification and according to [Holz1991] validation refers to all activities "used to check that the formal specification itself is logically consistent".

Formal verification is the process rigorously demonstrating that specified properties hold in a given system. However, there are computational complexities inherent to formal verification that makes it difficult to verify large designs. In order to formally prove properties of a system, formal models of the system, a formal way of expressing properties, and an algorithm for performing the check are needed. The Finite State Machines provide an example of these models.

However, formal methods can prove that an implementation satisfies a formal specification, but they cannot prove that a formal specification captures a user's intuitive informal understanding of a system. In other words, formal methods can be used to verify a system, but it is hard to validate a system [Vien1993]. The distinction is that validation shows that a product will satisfy its operational mission, while verification shows that each step in the development satisfies the requirements imposed by the previous steps.

The Finite State Machine (FSM) verification problem is to check the equivalence of two FSMs [KV1992]. The verification process of FSMs usually consists of deriving (either explicitly or implicitly) a product machine from the given FSMs, collapsing all the failure states into a failure state, and determining the reachability of the failure state from the initial state of the product machine.

Verification usually relies on state exploration techniques. Two classes of exploration algorithms have to be distinguished: exhaustive and non-exhaustive exploration algorithms [HK+2000].

Exhaustive validation performs an analysis of the complete model. The most common exhaustive exploration approach builds the reachability graph of a system by visiting all reachability states. The visited states in this graph are used during the exploration to avoid multiple visits of states. This ensures the termination of the algorithm. A reachability graph comprises all execution paths the system is able to perform. Deadlocks, livelocks or dead code can be discovered in this way. Due to the state space explosion problem, this method is only applicable to relatively small systems.

Non-exhaustive exploration algorithms cope with this state explosion problem by exploring only parts of a system. Certainly, this method cannot prove error-freeness for the whole system, but experience has shown that specification errors manifest themselves in many different states. Therefore, it is not necessary to cover all execution paths of the system to spot errors. The problem is to find a sufficient subset of all paths.

2.1.3 Test Derivation

It is essential to ensure that the approved standards and agreed specifications are exactly followed during the implementation process and to this end the product suppliers who implement the standards and specifications carry out testing during the product development cycle to assess conformance.

To demonstrate conformance to a standard or specification, a set of test events need to be executed in the form of data unit exchanges between a tester and the implementation under test. Sequences of test events are grouped into test cases and sets of test cases into test suites. A test case may not mean much if one does not know its purposes [WL1993]. The test purposes are statements indicating what kind of errors the test case tries to detect. Each test case usually corresponds to a specific test purpose reflecting a unique requirement in the specification of a test suite. It is normal that several thousand conformance test cases may be produced at different stages of development and executed against each implementation in a medium-size system. This is well beyond the capability of manual test production methods. However, with software tools, it is possible to produce error-free test suites from specifications such as in SDL without limiting the number of test cases and test coverage of the requirements [Monk1999].

Formal test derivation methods generally rely on the use of a mathematical model such as Labeled Transition Systems (LTSs) and Finite State Machines (FSMs) [PB1993]. Several formal description techniques have been standardized based on the two mentioned mathematical models; for example, LOTOS is based on LTS while SDL and Estelle are based on the FSM model. Much work on the test derivation from a given system specification has been done separately for the two models [PB1993]. However, these mathematical models are becoming inadequate when applied to systems of industrial size, where data are widely used [TR1999]. In a practical situation, for example in a distributed system, a protocol specification includes variables and operations based on the variables; using pure "FSM" will be much complicated. A well-known compact model, called EFSM (Extended Finite State Machine), see, e.g., [PB+1999] allows us to model protocols in a succinct way. The EFSM model is based on the FSM model but is extended with data and related concepts such as (parameters, variables, operation on data, predicates on data, etc.). The EFSM model is the basis of the semantic representation of Formal Description Techniques such as SDL. The concepts of FSM and EFSM model will be formally given in the following Section 2.2. The concepts of SDL and the related tool will be discussed in Section 2.3 and Section 2.4.

2.2 Definitions of FSM and EFSM Models

2.2.1 The FSM Model

In this section we give some important definitions and notations of finite state machines that will be frequently used in the subsequent sections. The model of FSM considered here is based on the so-called Mealy machine. The following definitions and notations are taken from [HU1979, Gill1962, Star1972, PY+1996].

Definition 2.1 (FSM):

A *finite state machine* (FSM) M is a 6-tuple (X, Y, S, s_1, h, D_A), where

- X is a finite set of input symbols;
- Y is a finite set of output symbols;
- S is a finite set of states;
- s_1 is the initial state;
- $D_A \subseteq S \times X$ is a specification domain;
- h is a behavior function $D_A \rightarrow P(S \times Y)$, where $P(S \times Y)$ is the powerset of $S \times Y$.

The behavior function h characterises possible transitions of the machine. A specified transition from state s_i to s_j with input x and output y can be represented in the form s_i-x/y->s_j. Usually, s_i is called the *head* (or initial) state of a transition and s_j is called the *tail* (or final) state of a transition.

The machine M is deterministic (FSM) if for all $(s, x) \in D_A$, $|h(s, x)| = 1$. In the case of deterministic FSMs, instead of the behavior function h, we use two functions: the *transfer* (or the next state) function δ and the *output* function λ to characterise the behavior of the finite state machine.

Definition 2.2 (completely defined and partially defined machine):

A FSM M is said to be:

- *completely defined* (or specified) if $D_A = S \times X$. It means that the behavior function h is defined (or specified) for all the state-input combinations.
- *partially defined* (or specified) if $D_A \subset S \times X$. It means that there should be some state-input combinations for which the behavior function h is not defined.

We will mainly consider completely defined machines in this work. So if not mentioned explicitly, all the machines are completely defined.

Definition 2.3 (p-equivalent, equivalent states, and equivalent FSMs):

State s of the machine M_1 and state r of the machine M_2 are said to be:

p-equivalent if for any input sequence of length p, the sets of output sequences produced by the machine M_1 in state s and the machine M_2 in r in response to the input sequence coincide. If s and r are *p-equivalent* for any p, then they are *equivalent.* If s and r are not equivalent, they are said to be *distinguishable.* Two complete FSMs are said to be equivalent if their initial states are equivalent.

Definition 2.4 (reduced and minimal machine):

A FSM M is said to be *reduced* if all its states are pairwise distinguishable. A FSM M is said to be *minimal* if the number of states of M is less than or equal to the number of states of any machine M_1 equivalent to M.

A minimal machine is reduced, but the converse is not true for partially defined machines.

Definition 2.5 (projection):

The h^2 is projection of h, where $h^2(s, \alpha) = \{\beta | \exists s' \in S\ [(s', \beta) \in h(s, \alpha)]\}$, for all $\alpha \in X^*$.

The set $h^2(s, \alpha)$ contains all output sequences that can be produced by the FSM in response to the input sequence α applied in the state s.

Definition 2.6 (reduction relation):

Given two FSMs $M_1 = (X, Y, S, s_1, h)$ and FSM $M_2 = (X, Y, T, t_1, H)$, states $s \in S$ and $t \in T$, state t is said to be a *reduction* of state s, written $t \leq s$, if, for all input sequences $\alpha \in X^*$, the condition $H^2(t, \alpha) \subseteq h^2(s, \alpha)$ holds; otherwise t is not a reduction of s.

Note that $t \leq s$ and $s \leq t$ says that s and t are equivalent states.

The FSM model is the basis for a more powerful model of extended finite state machines.

2.2.2 The EFSM Model

Similar to FSM, an EFSM contains a set of states and several transitions from one state to another, but the data part of an EFSM transition includes input/output parameters, context variables, and a transition firing enabling condition. The difference between an EFSM and an FSM is that an EFSM associates each transition not only with input and output actions but also with assignment action and condition [WL1993].

The model of a Mealy (finite state) machine extended with input and output parameters, context variables, operations and predicates defined over context variables and input parameters can be formally defined as follows [PB+1999].

Definition 2.7 (EFSM):

An *extended* finite state machine (EFSM) M is a pair (S, T) of a finite set of states S and a finite set of transitions T between states from S, such that each transition $t \in T$ is a 7-tuple (s, x, P, op, y, up, s'), where

- $s, s' \in S$ are the initial and final states of the transition, respectively;
- $x \in \mathrm{X}$ is input, X is a set of inputs, and D_{inp_x} is the set of input vectors, each component of an input vector corresponds to an input parameter associated with x;
- $y \in Y$ is output, Y is a set of outputs, and D_{out_y} is the set of output vectors, each component of an output vector corresponds to an output parameter associated with y;
- P, op, and up are functions, defined over input parameters and context variables V, namely,
  - P: $D_{inp_x} \times D_V \rightarrow \{True, False\}$ is a predicate, where D_V is a set of context vectors $\vec{v}$;
  - op: $D_{inp_x} \times D_V \rightarrow D_{out_y}$ is an output parameter function;
  - up: $D_{inp_x} \times D_V \rightarrow D_V$ is a context update function.

The function P, op, and up define the dynamic properties or the behavior of the EFSM. An EFSM machine receives input along with input parameters (if any) and computes the predicates that are satisfied. One transition among those labeled with the received input and a satisfied predicate is triggered and the EFSM moves from one state to another state. At the same time, it produces an output along with output parameters and changes the values of its variables according to the update function.

Specifically, we normally use (s—x, P/op, y, up→s') to denote a transition $t \in T$. If, in t, P is a True constant, P can be dropped from the transition. Similarly, when the transition does not change the values of the variables, the update function *up* can be omitted. Also, the output parameter function can only be absent when output y has no output parameters at all. Notations (s—x, P/y→s'), (s—x/y, up→s'), (s—x/y→s') are examples of notations used for such situations.

For pragmatic reasons we assume that update function is typically represented by a sequence of assignments of new values to context variables and parameters, so it is easy to determine which variables define new values for variables and output parameters. Typically, in scope of a transition, each variable or output parameter depends only on few values. Therefore, in most examples a slightly different notation, with assignments of variables and parameters in the SDL style will be used.

In Definition 2.8, we present the main properties of the EFSM [PB+1999] that will be used in the following chapters.

Definition 2.8 (consistent, completely specified, deterministic, observable machines):

An EFSM M is said to be:

- *Consistent* if for each transition t, every element in $D_{inp_x} \times D_V$ evaluates exactly one predicate to True among all predicates guarding transitions with the start state and the input of t; in other words, the predicates are mutually exclusive and their disjunction evaluates to True.
- *Deterministic* if any two transitions outgoing from the same state with the same input have different predicates.
- *Observable* if for each state and each input, every outgoing transition with the same input has a distinct output.

An EFSM exhibits a much richer behavior than a simple finite state machine. A simple finite state machine equivalent to an extended machine with only a few variables or parameters would be too large and complex to be constructed and/or understandable, whereas the behavior of the extended machine can be easily followed. An extended finite state machine can also represent the behavior that is impossible for a finite machine if some variables or parameters have an infinite number of values, for example, if a parameter is an integer. In this case no FSM equivalent to the given EFSM can be constructed.

2.3 An Overview of SDL Language

The specification and description language (SDL) is an object-oriented, formal language defined by The International Telecommunications Union–Telecommunications Standardization Sector (ITU–T) as recommendation Z.100 [CCITT92].

SDL has a number of advantages compared to other high-level languages and traditional low-level languages such as C, C++, or Java. SDL has a rich grammar that describes behavior and is unambiguous. Therefore, it is possible to build tools for the simulation of SDL systems and for the validation of formal characteristics, like deadlock avoidance. In short, this means that errors can be detected at a very early stage.

The language SDL is intended for the formal specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals. It is especially well suited for specification of communications protocols, reactive systems such as switches and routers and distributed systems. SDL has been designed for the specification and description of the behavior of such systems, i.e., the interworking of the system and its environment. It is also intended for the description of the internal structure of a system, so that the system can be developed one part at a time.

Figure 2.1 shows four main hierarchical levels of an SDL system. In an SDL specification, a *system* consists of a number of blocks connected by channels. A *block* is an enclosure for further structuring of the system. Channels connect the blocks with each other and with the environment of the system in one-way or two-way mode. A block consists of processes connected by signal routes. Each process is an extended finite state machine (EFSM). These machines (or processes) run in parallel. They are independent of each other and communicate with discrete messages, called *signals*. A process can also send signals to and receive signals from the environment of the system. The behavior of a state machine is characterized by a set of transitions. A transition to another state or the same state occurs whenever a stimulus (or input) is consumed. When a process is in a state it accepts stimuli from its input port. These stimuli can be signals received by the input port or timers. When a process enters a new state, it means that a transition terminates. EFSM enables decisions to be made in transitions based on the value associated with a variable so that the state which follows when a specific input is consumed is not only determined by the existing state and input. A transition may contain the following actions:

- *Output:* to send signals.
- *Task:* to change the value of variables. Local variables for each machine may hold details about the history of the machine.
- *Create:* to create process instance.
- *Decision:* to split into several sequences of actions.
- *Call:* to activate procedures. A procedure is a parameterised part of a process with its own scope.
- *Set, reset:* to manipulate timers.

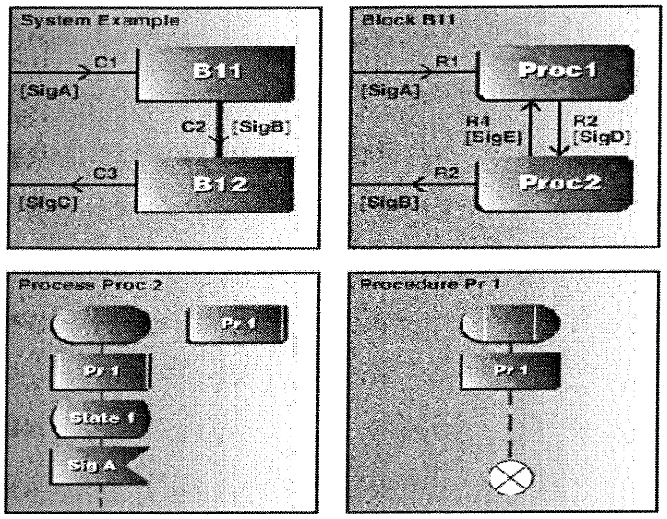

Figure 2.1 Four Main Hierarchical Levels of an SDL System

The SDL language supports two equivalent notations: the graphical notation (SDL–GR) and the textual notation (SDL–PR). The graphical notation (SDL–GR) is a standardized graphical representation of the system. SDL elements such as system, block, process, signal etc. are drawn using standardized graphical symbols. The textual notation (SDL–PR) is a textual phrase representation of the SDL system, or in other words, it is a SDL "source code" [BH1989].

2.4 An Overview of the ObjectGeode Tools

Supporting simulation, validation, code generation and test generation is the most important reason for using SDL as a system specification language. Since SDL is graphical and formal, it is possible to use tools to automate simulation, verification and validation. Automation is not possible with non-formal notations and languages, and with it designers can quickly determine the completeness and correctness of the specifications early in the development process. SDL is accepted by the industry, because it has a good maintenance support. SDL commercial tools, such as SDT and ObjectGeode, provide integrated graphical environments for developing of SDL systems. In this work, we use the ObjectGeode environment to implement and check our theoretical results.

ObjectGeode [GEODE-1] is a toolset dedicated to analysis, design, verification and validation through simulation, code generation and testing of real-time and distributed applications. Such applications are mainly used in fields such as telecommunications, aerospace, automotive, process control or medical systems etc.

ObjectGeode is an advanced integrated environment for the development of distributed real-time systems. The ObjectGeode environment consists of highly advanced graphical tools. It provides graphical editors, a powerful simulator, and a C code generator targeting popular real-time OS and network protocols, and a design-level debugger. Complete traceability is ensured from requirements to code. It also supports a coherent integration of complementary object-oriented and real-time approaches based on the UML, SDL and MSC standards languages. ObjectGeode has the following components [GEODE-1].

(1) Graphical Editor for creating and editing of the SDL system

Various editors provide for intuitive means of creating, modifying, and viewing the diagrams of an ObjectGeode description: Architecture, Communication, State machines and Message Sequence diagrams. Consistency and compliance with notation rules is controlled by the Checker. Powerful multi-user features are available for large/distributed projects.

(2) Simulator that simulates the system in the graphical environment

The powerful simulation tools integrated in ObjectGeode simulator can provide visual design level debugging, verification and validation techniques. They also can detect modeling errors before coding start, and show proof that the model complies with requirements. The ObjectGeode simulator provides three simulation modes: interactive, random, and exhaustive [GEODE-3].

- The *interactive* or *step by step* mode provides a fine-grained simulation. The user is free to decide which parts of the design that will execute, and to move up and down the simulation process with Undo and Redo commands. Like conventional debuggers, the simulator offers a graphical view of design being executed, indicates the current position in the corresponding MSC, and produces results in real-time.

- The *random* mode derives a pattern from a number of patterns provided by the developer to explore some of the possible application behavior.

- The *exhaustive* mode requires the simulator to explore all behavioral paths. When running in this mode, the simulator checks all verification properties. If a violation is detected, a scenario is created, which reflects how to get the faulty condition.

These three modes may be alternatively used during the same simulation session. For each of these modes, the simulator generates scenarios containing the results of the verification (deadlock detection etc). These scenarios can be replayed in interactive mode and expressed graphically in the form of Message Sequence Charts (MSCs).

(3) TestComposer & TTCN Test Suite Publisher

TestComposer [KJ+1999] is an automated test generation tool for conformance testing from SDL and MSC models. TestComposer takes as inputs a SDL specification, a specification of the test environment and a possibly empty set of user defined test purposes. These user defined test purposes can be built interactively using the simulator, or written using MSC or GOAL observers. The tool then completes the set of test purposes by computation of new test purposes according to structural coverage. Then the generation engine is fed with each test purpose, and produces test cases, which are stored in the test case database. Finally, through the test case database API, the test suite is built and written in a TTCN-mp file. TTCN Test Suite Publisher, extension to Test Composer, enables to translate the generated test suites into TTCN format for Telelogic Tau TTCN Suite.

(4) Code Generator

Fully executable C code of the (distributed) multi-task real-time application is automatically generated. Makefiles are also generated to automate the building process. Through dedicated ObjectGeode Run-Time Libraries, the generated code is then mapped on to real-time operating systems such as CHORUS, Nucleus, OSE, OSEK, pSOS+, VRTXsa ®, VxWorks, WIN32 or various flavors of UNIX and network protocols such as TCP/IP. The DesignTracer allows the current design description to be visualized interactively and trace information (including time) to be displayed as MSC diagrams.

2.5 Conclusion

Until now we have reviewed the related theory of formal methods and the concepts of the finite state machine and extended finite state machine. The EFSM model is the basis of the semantic representation of FDTs such as SDL. The language SDL is an object-oriented, formal language defined by ITU–T as recommendation Z.100, which is intended for the formal specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals. The toolset ObjectGeode is dedicated to analysis, design, verification and validation through simulation, code generation and testing of real-time and distributed applications. In the next chapter, we discuss the ideas for state abstraction techniques for EFSM.

Chapter 3

State Abstraction Techniques for EFSM

3.1 The Need for State Abstraction

Most of model checkers used to verify finite state systems perform worse when the number of states in the modeled system increases. This is a notorious state explosion phenomenon. In order to alleviate the state explosion problem some states of an FSM underlying the system can be merged. States that were different now become identified so that the resulting machine with fewer states can be considered as an abstraction of the given machine. The fewer states a formal specification has, the easier all the problems of verification and test derivation (and related problems) should be. The problem of state abstraction can be solved in various ways that differ in the nondeterminism of resulting specifications. Our goal is to elaborate state merging techniques such that

(1) The resulting specification presents the behavior of the original specification.

(2) The resulting specification is observable (its underlying EFSM is observable).

(3) State merging can be performed on the SDL specification.

3.2 Existing Abstraction Techniques for the FSM Model

According to [Oiko1996], an abstraction of a finite state machine (FSM) M consists in lumping (aggregating) some of its states, inputs, and outputs into classes, which then become the states, inputs, and outputs of a smaller FSM M_A. Even if M is deterministic, the abstracted machine M_A will, in general, be nondeterministic. In spite of nondeterminism introduced by abstraction, such an abstraction can still be useful. For example, in the testing context, there is still a class of faults in the system which are immediately-detectable upon occurrence using an abstracted machine instead of the original one [Oiko1996]. [Oiko1996] presents some criteria for selecting an optimal abstraction of a given finite state machine. It also gives an algorithm which computes an approximately optimal abstraction in reasonable time. This work concentrates only on FSM abstraction, while to abstract an SDL specification, one has to deal with EFSMs.

[GS+1996] distinguishes three classes of reduction mechanisms, heuristics, partial order simulation methods, and optimization strategies, which can be used for handling the complexity of SDL specifications. Heuristics are based on the assumptions about the behavior of the system to be tested, or its environment. They avoid the elaboration of system traces which are not in accordance with the selected assumptions. Partial order simulation methods avoid complexity which is caused by an interleaving semantics of the specification language. They intend to limit the exploration of traces for concurrent executions. Optimization strategies intend to reduce the possible behavior of the system environment. [GS+1996] also describes the way they work. In [GS+1996], the entire behavior of an SDL system is treated as a labeled transition system (LTS) and is described in form of a behavior tree.

[LG+1995] studies the property preserving transformations for reactive systems. A key idea of [LG+1995] is the use of Galois connection and (α, γ)-simulation which is the same as the standard simulation often used to define implementation.

Furthermore, (α, γ)-simulation induce abstract interpretations and this allows to apply an existing powerful theory for program analysis.

The method for machine reduction by successive merging equivalent states was given in [Gill1962]. This method emphasized on merging simply equivalent pair of states successively to get a reduced machine of the original one. State minimization for the EFSM model is a much harder problem, for which no efficient technique exists. Moreover, we are not aware of any work done on state abstraction for EFSM or SDL. Our main goal here is to offer techniques for state abstraction that can be used not only for EFSM, but also for SDL specifications.

3.3 Abstracting States in the FSM Model

3.3.1 Basic model of abstraction

In this section we will give some definitions and notations needed to define state abstraction of an FSM.

Definition 3.1 (abstracted machine):

Let A be an observable completely defined FSM with the set of states S and $\pi = \{B_1, B_2, \ldots\}$ be a partition on S. An FSM over the same alphabets is called a *factor* FSM for A, denoted A_π, if each state of A_π is a block of the partition π and, for each transition $(s\text{—}x/y\rightarrow s')$ of A, there exists a transition $(\pi(s)\text{—}x/y\rightarrow\pi(s'))$ and vice versa.

Here $\pi(s)$ denotes the block which contains s. A state $\pi(s)$ of the factor machine is also called a *factor* or *image* of the state s. $(\pi(s)\text{—}x/y\rightarrow\pi(s'))$ is called a *factor* or *image* of the transition $(s\text{—}x/y\rightarrow s')$. A factor machine is also called an *abstracted* machine, also known as an *abstraction*.

Example:

An example of FSM *A* is shown in Figure 3.1. It has the state set $S = \{1, 2, 3, 4\}$. The machine FSM *A* is completely specified, deterministic, and thus observable.

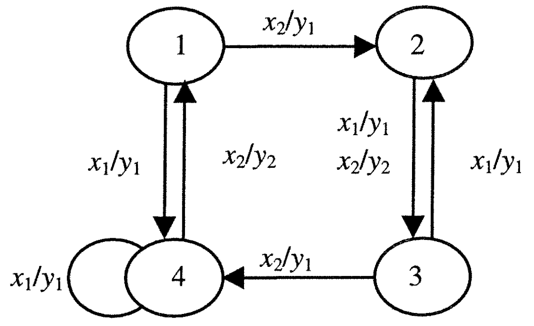

Figure 3.1 An FSM

For the machine *A*, we may have partitions of *S* such as $\{\{1, 3\}, \{2, 4\}\}$, $\{\{1, 4\}, \{2, 3\}\}$, $\{\{1\}, \{2, 3, 4\}\}$, $\{\{1, 2, 3, 4\}\}$, $\{\{1\}, \{2\}, \{3\}, \{4\}\}$.

For the partition $\pi = \{\{1,3\}, \{2, 4\}\}$, the abstracted machine of *A* can be built as shown in Figure 3.2.

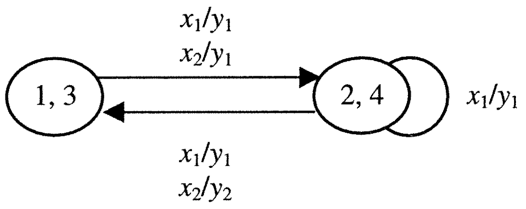

Figure 3.2 A Non-observable Abstracted Machine

Here we obtain a non-observable abstracted FSM of *A* because from the state $\{2, 4\}$ we have two transitions with the same input-output label x_1/y_1 which lead the FSM into two different states: one to the state $\{2, 4\}$, another to the state $\{1, 3\}$.

For the partition $\pi = \{\{1,4\}, \{2, 3\}\}$, we can construct the abstracted machine of A in Figure 3.3, it is observable, because for each state and each input, every outgoing transition with the same input has a distinct output.

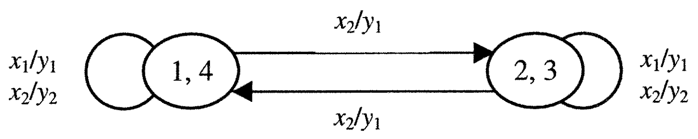

Figure 3.3 An Observable Abstracted Machine

3.3.2 Behavior Preservation by an Abstraction

The following statement says that the abstracted machine preserves the behavior of the original machine (while a new behavior also appears).

Let S_A and S_B be state sets of FSMs A and B, respectively. A mapping φ of S_A into S_B is called a (transition) *homomorphism* from A into B if for each transition (s_1—x/y→s_2) of A there exists a transition ($\varphi(s_1)$—x/y→$\varphi(s_2)$) of B. When for each transition (s_1^B—x/y→s_2^B) of B there exists a transition (s_1—x/y→s_2) of A such that $\varphi(s_1) = s_1^B$ and $\varphi(s_2) = s_2^B$, φ is called a homomorphism from A *onto* B.

Proposition 3.1

The machine B is isomorphic to an abstraction of A iff there exists a homomorphism from A onto B.

Let (A, s) denotes FSM A initialized into the state s.

Proposition 3.2

(A, s) is a reduction of (A_π, $\pi(s)$).

Proof. Clearly the partition π induces a homomorphism from (A, s) onto $(A, \pi(s))$. It is easy to prove by induction that for any input sequence every output sequence produced by FSM A in state s can also be produced by the FSM A_π in the state $\pi(s)$.

This means that every input-output sequence produced by (A, s) is also produced by $(A_\pi, \pi(s))$. Note that $(A_\pi, \pi(s))$ may produce input-output sequences, which (A, s) can not. Clearly, the abstracted machines in Figure 3.2 and 3.3 preserve all behaviors of their original machine, but the abstracted machines exhibit behaviors that the original machine does not. For our example in Figure 3.3, the abstracted machine has the nondeterministic behavior that the original machine does not have: $\{2, 3\} \rightarrow x_2/y_1 \rightarrow \{1, 4\}$ or $\{2, 3\} \rightarrow x_2/y_2 \rightarrow \{2, 3\}$ and $\{1, 4\} \rightarrow x_2/y_1 \rightarrow \{1, 4\}$ or $\{1, 4\} \rightarrow x_2/y_2 \rightarrow \{2, 3\}$.

3.3.3 Observabilization

The factor machine A_π is not necessary observable, as, for example, in Figure 3.2. Obviously, it is observable only when any two transitions of A that start from one block of π with the same input/output label converge to a same block.

Proposition 3.3

A_π is observable iff all transitions of A with the start states in the same block and with the same input/output label lead to the states of a same block of the partition π.

The goal of abstraction is not just to reduce the cardinality of the state set but to obtain a simpler machine that is easier to handle. However, a non-observable machine with fewer states than the observable one may be more compact but is not necessary easier to process. The problem is that an observable machine may be interpreted as a deterministic automaton (acceptor), see [Star1972], while a non-observable one cannot, and most of the effective methods in automata theory and testing theory are developed either for deterministic automata or for observable machines.

Even when a non-observable machine is simpler than the original observable one, there exists certain lack of tools which may be used to work with these machines. For example, we are unaware of any model checker that is capable of finding of input sequences that allows one to deterministically [PY+1996] reach a given state from the initial state.

Therefore, in many applications, an observable equivalent of a non-observable factor machine is preferable, so the question arises as to how one can obtain an observable form of an abstracted machine, preserving at least some elements of the behavior of the original machine. There exists a well-known algorithm of determinization of an automaton, which can be used to obtain an equivalent observable abstracted machine from the factor machine, see, e.g., [HU1979, Star1972]. The problem of this solution is that, in many cases, determinization gives a rise (exponential, in the worst case) in the number of states, so the obtained machine with more states than in the abstracted machine can hardly be considered as a simplification of the original machine. Moreover, this method may trigger a state explosion that can be undesirable. Therefore, we consider methods for transforming a given non-observable FSM into another abstracted observable machine such that do not increase the number of states. The resulting machine should either be equivalent to the factor machine or preserve its behavior as much as possible.

In the following paragraphs, three techniques which can be used in the state abstraction of FSM are presented. The first one uses a coarser partition than originally given. The second one uses an additional "trap" state. The last one redirects certain transitions.

3.3.3.1 Use of a Coarser Partition

Proposition 3.4

There exists a partition $\pi' \geq \pi$ such that A_π' is observable.

Proof. $\{S\}$ is such a partition.

It is clear from Propositions 3.1 and 3.2 that by consecutive merging state blocks in which A_π can go from a block via transitions with the same input-output label, we will obtain the minimal partition $\pi' \geq \pi$ such that $A_{\pi'}$ is observable. Such a minimal $\pi' \geq \pi$ is unique for each A_π. However, this procedure of implied merging of states may easily collapse all the states into a single state. In our example, Figure 3.4 shows the observable machine, which is constructed from the abstracted machine in Figure 3.3.

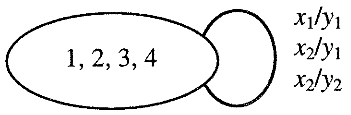

Figure 3.4 An Observable Abstracted Machine Using a Coarser Partition

As this example shows, the resulting machine might be too "chaotic", the observability is obtained at a cost of "over-abstracting" the original machine. So we have to try another approach.

3.3.3.2 Using a Trap State

A nondeterministic FSM is non-observable if it has several transitions that share common input-output label, starting state, but have different ending states. We call them *conflicting* transitions. One can get rid of non-observability of A_π without further state merging, simply by deleting all conflicting transitions. The obtained FSM becomes partially defined. Here a partially defined FSM, unlike the SDL process, is not understood as a machine that discards some signals. We assume that consuming an undefined input signal a partially defined FSM can exhibit any behavior. Such a partially defined FSM can be modeled by a completely defined (nondeterministic) FSM, which, for each state s of the partially defined FSM and each undefined in state s input x, has transitions from s into a special trap "chaos" state with the input label x and all possible outputs. We call the transitions leading to the trap state *trapping* transitions. The trap state has self-looping (trapping) transitions with all possible input-output labels.

The use of a trap state allows us to obtain a conservative abstraction of the original machine, as in the case of the implied merging of states. Naturally, when "too many" transitions are becoming trapping the abstracted machine may converge to a "chaos" machine, as in the approach based on implied merging of states. At the same time, a model checker exploring an abstraction with the trap state may be instructed to abandon a path with a trapping transition. Intuitively, this method is preferable, even if it is not guaranteed to always deliver better results than the approach based on the implied merging of states.

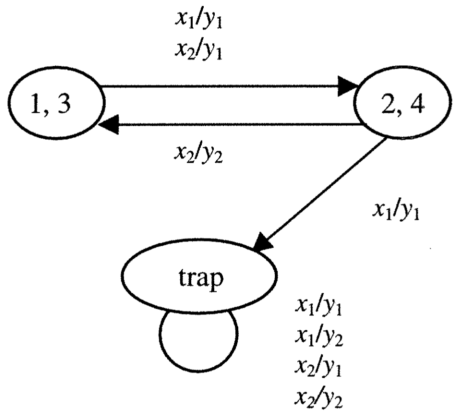

Figure 3.5 An Abstracted Machine with a Trap State

3.3.3.3 Redirecting Conflicting Transitions

Instead of replacing conflicting transitions by trapping transitions, we can resolve a conflict by keeping just one transition among the conflicting ones for each state. Let obs(A_π) be a machine obtained from A_π by keeping one transition among all conflicting transitions, i.e., the transitions that share the same starting state and input-output label, and erasing all the others. Speaking more precisely, we redirect all the conflicting transitions into one state that is the ending state of one transition among them, for example, we can choose state (2, 4) in Figure 3.2 and redirect the two conflicting transitions to this state as shown in Figure 3.6.

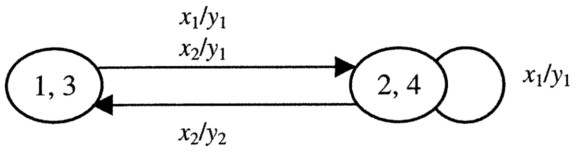

Figure 3.6 Redirecting All the Conflicting Transitions into One State

By doing this, we extract, in fact, a maximal observable submachine of the factor machine. Clearly, such transformation alters the behavior of the factor machine, unless only equivalent states were merged.

We can try to identify parts of the behavior of the original machine that are preserved in obs(A_π). For a given partition π we calculate a parameter p such that for each pair of transitions ($\pi(s_1)$—x/y→$\pi(s_2)$) of obs(A_π) and (s_1—x/y→s_3) of the original machine the states s_2 and s_3 are p-equivalent, $p \geq 0$. As usually, we assume that every state produces an input-output sequence ε of length 0. Therefore, any states are 0-equivalent. The value of the parameter p characterizes indistinguishability of ending states of conflicting transitions. Final states of any conflicting transitions cannot be distinguished by any input sequence of length less than or equal to p.

The next proposition shows that the p-behavior of every state of A is preserved in a corresponding state of obs(A_π). The *p-behavior* of state s of the machine M is the set of all input-output sequences of length p produced by this state s of M.

Proposition 3.5

The p-behavior of a state s of A is a subset of the p-behavior of the image state $\pi(s)$ of obs(A_π). The subset is proper unless certain blocks of π contain only p-equivalent states.

Now we establish how the (p+1)-behavior of A is affected by the proposed transformation.

Proposition 3.6

Let (s_1—x/y→s_2) be a transition of the original machine A and ($\pi(s_1)$—x/y→$\pi(s_2)$) be the corresponding image transition of obs(A_π). Then the set of all input-output sequences of length (p + 1) produced by state s_1 that start with x/y is a subset of the set of such sequences of the image state $\pi(s_1)$. It is a proper subset unless certain blocks of π contain only p-equivalent states.

The following statement gives a sufficient condition for the preservation of the (p + 1) behavior of a state of the original machine by the image state. The statement shows that a certain choice of transitions preserves in obs(A_π) the (p + 1)-behavior of at least one state of the original machine in a block.

Corollary 3.7

Let s_1 be a state of A and for every outgoing transition (s_1—x/y→s_2) of s_1 there exists an image transition ($\pi(s_1)$—x/y→$\pi(s_2)$) of obs(A_π). Then (p + 1)-behavior of s_1 is a subset of (p+1)-behavior of $\pi(s_1)$, i.e., $s_1 \leq_X^{p+1} \pi(s_1)$.

If states of the FSM A are decorated with priorities (reflecting, for instance, their importance), according to Corollary 3.7, it makes sense to keep in obs(A_π) transitions which correspond to transitions of a state with a higher priority (importance).

The following statement asserts that the (p + 2)-behavior may be lost in obs(A_π).

<u>Proposition 3.8</u>

It is not always possible to find an image state of obs(A_π) which preserves elements of the (p + 2)-behavior of a state of the original machine.

As mentioned in Chapter 2, the finite state machine in SDL is an extended finite state machine. The problem of state abstraction becomes more involved for this type of machines than for pure FSMs. The problem is how to treat input /output parameters, variables and decisions appeared in the transitions of EFSM. In the following section, we discuss techniques for abstracting states in an EFSM.

3.4 Abstracting States in the EFSM Model

The techniques developed in Section 3.3 help us address the problem of state abstraction in EFSM. The state merging abstraction procedure is quite straightforward, it extends from FSM onto EFSMs directly. Therefore, we feel that there is no need to formally define it. We believe that here illustrating it on an example EFSM in Figure 3.7 (which is borrowed from [PB+1999] with some simplifications) will be sufficient.

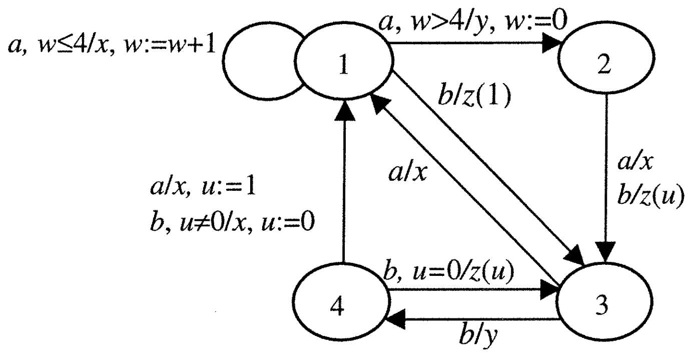

Figure 3.7 The EFSM *M*

The EFSM on Figure 3.7 has four states, two integer context variables, two inputs *a* and *b*, three outputs *x*, *y*, *z*, the latter is parameterized with an integer parameter, four transitions are guarded with predicates different from True. Figure 3.8 represents an abstraction of this EFSM defined by the partition {{1, 3}, {2, 4}}. Another abstraction is given in Figure 3.10, it is defined by the partition {{1, 4}, {2, 3}}. Similar to the FSM case, an abstracted EFSM may become non-observable. The observabilization of EFSMs will be discussed in the following section.

3.5 Observabilization of EFSM

There are two types of conflicting transitions in non-observable EFSM. First, all conflicting transitions of a state with a given input/output label lead into one state. Second, conflicting transitions of a state which have a given input/output label lead to different states. The former case was not possible for FSM. An example machine with conflicting transitions is depicted in Figure 3.8. Indeed, state (2, 4) has two transitions, leading into state (1, 3) with input *a* and output *x*. Moreover, the state (2, 4) has two transitions, leading into (1, 3), with the label *b*/*z*. The state (1,3) has two loops labeled with the input *a* and output *x*.

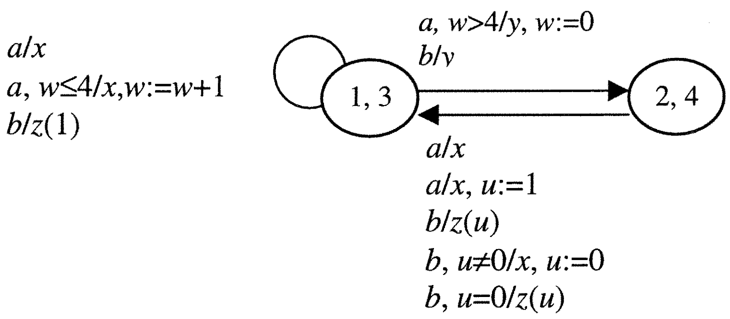

Figure 3.8 An Abstracted EFSM

As to this type of conflicting transitions (with the same ending state) we suggest to merge them into a single transition. The predicate of the obtained transition is the disjunction of the predicates of merged transitions. The context variables that are updated in different ways on merged transitions should be removed (abstracted) from the given EFSM along with all assignments and predicates that use them, as it is proposed in [PB+1999]. Conservative abstraction of context variables in EFSM is by itself an independent problem. The paper [PB+1999] suggests that, to make an abstraction of context variables conservative one should delete the set of variables closed under the dependency relation. The above work does formally define the dependency relation. So we detail this concept here, mainly based on the results of [Wang2001].

Typically, variable v_i of function $f(v_1,...,v_r)$ is called *essential* if there exists at least two r-tuples of values of variables of the function f that differ only in value of this variable but deliver different values of this function. Usually, one say that a value of a function of many variables depends on a particular variable v if this variable is essential. Therefore, a predicate depends on a variable if the latter is an essential variable of the predicate function. A variable v *locally depends* on a variable u if in the scope of the transition a new value of the former depends on a value the latter (or in other words, u is an essential variable of the projection of the update function onto the variable v). The transitive closure of this relation is called the *dependence*. The transitive closure is the minimum transitive relation that includes the given one. Since the number of variables in EFSM is finite, v depends on u iff there exists a sequence of variables that starts from v and ends with u, and each element of which locally depends on the right-hand neighbor. Determining essential variables may be a complex problem. But usually, in practice, update functions are represented by sequences of assignments. A variable v is said to *directly depend* on a variable u if there exists an assignment with the variable v in the left-hand part, and the variable u in the right. The transitive closure of the *direct dependence* is a conservative approximation of the dependence relation.

Removing a variable also implies replacing all the predicates that depends on this variable with the True predicate. Moreover, all output parameters that depend on the removed variables, should also be abstracted. We follow this approach except for output parameters. Complete removal of an output parameter in all appearances of the corresponding output in transitions that depends on a context variable to be abstracted is in a sense too "aggressive" abstraction of EFSM, because the parameter will be removed completely from the machine even if it depends on the removed variables only in a single transition. Therefore, we follow the output parameter abstraction approach developed in [Wang2001]. Namely, in the abstracted machine we extend the domain of each output parameter with a designated value '*', which means that this parameter can take any value. Thus, instead of removing an output parameter, we assign this value to a parameter on each transition that depends on deleted variables. An example of such observabilization of the EFSM in Figure 3.8 is presented in Figure 3.9.

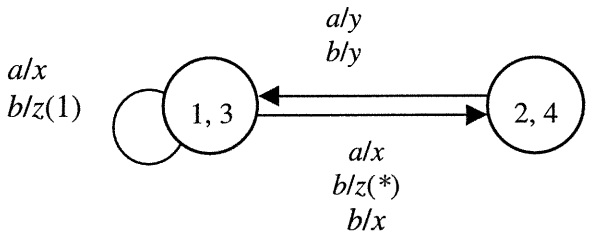

Figure 3.9 An Abstracted Observable EFSM

The behavior of the obtained machine differs from that of the original machine; formally speaking, the original machine is not even a reduction of the abstraction since the abstraction uses the parameter value * that is not present in the original machine. Therefore, we introduce a quasi reduction relation.

An EFSM is called a *quasi reduction* of another EFSM if for each input sequence, a possible parameterized output of the former machine is a subset of possible reactions of the latter after substitution of * parameters values with all possible combinations of other values of these parameters.

The original machine is a quasi reduction of the obtained observable machine. In the case of parameters with finite domains, replacing each transition with star output parameter values by a number of transitions that together provide all the possible combinations of the values of the parameters from the original domains, while each transition uses only one such a combination, would result in an EFSM such that the original EFSM becomes its reduction.

The machine obtained in our example (Figure 9) is much simpler, therefore easier to handle than the original machine.

So far, we have discussed conflicting transitions leading to the same ending state. As to conflicting transitions that lead to different ending states, the above approach is not applicable. To remove those we adapt a technique based on using a trap state. Namely, all these transitions are redirected to a trap state (and thus merged), where output parameters are instantiated with the value *. As with the previous type of conflicting transitions, the predicate of the obtained transition is the disjunction of predicates of merged transitions. The behavior of the EFSM does not depend on context variables in the trap state. Therefore, the update function may be arbitrary. Consider the example EFSM in Figure 3.10.

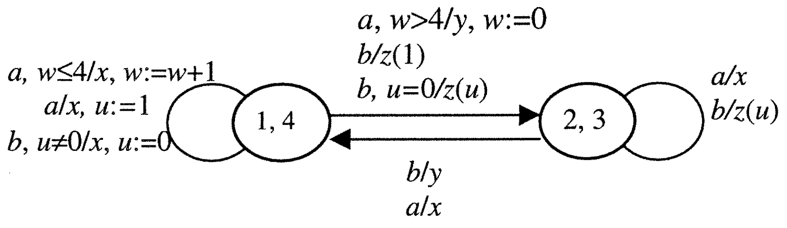

Figure 3.10 An Abstracted EFSM

The EFSM has both types of conflicting transitions. The conflicting transitions ($s_{1,4}$—a, w≤4/x, w:=w+1→$s_{1,4}$) and ($s_{1,4}$—a/x, u:=1→$s_{1,4}$) lead into the same state and therefore have to be merged. Variables w, u, as well as the dependant and predicates are abstracted from the EFSM. Similarly, the conflicting transitions ($s_{1,4}$—b/z(1)→$s_{2,3}$) and ($s_{1,4}$—b, $u = 0$/z(u)→$s_{2,3}$) are merged into ($s_{1,4}$—b/z(*)→$s_{2,3}$). The conflicting transitions ($s_{2,3}$—a/x→$s_{2,3}$) and ($s_{2,3}$—a/x→$s_{1,4}$) are merged into ($s_{2,3}$—a/x→*trap*). This gives an observable machine represented in Figure 3.11.

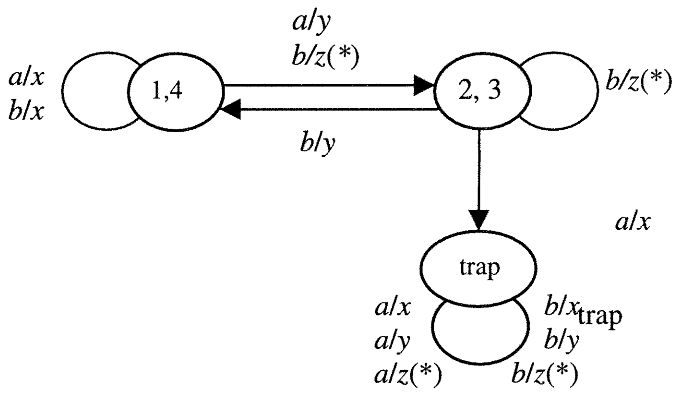

Figure 3.11 An EFSM with Trap State

It can be proven that after observabilization with the above method we obtain a machine such that the original machine is a quasi reduction of the former.

There are cases when such observabilization may be performed without altering the behavior and it may lead to the simplification of EFSM. Namely, this is the case when few conflicting transitions of a state share the same context update and output parameter functions.

Let these transitions be

$(s\text{–}x,P_1/op,y,up\rightarrow t)$,

$(s\text{–}x,P_2/op,y,up\rightarrow t)$,

…

$(s\text{–}x, P_n/op, y, up\rightarrow t)$

then these conflicting transitions could be safely merged into

$(t\text{–}x, P_1\vee P_2\vee\ldots\vee P_n/op, y, up\rightarrow t)$

Without any changes of the input output behavior, therefore whenever such transitions are detected, they should be merged.

3.6 Conclusion

In order to alleviate the state explosion problem some states of an FSM underlying the system can be merged. Three techniques can be used in the state abstraction of FSM and EFSM with respect to the preservation of the observability and behavior of the initial machine. The first technique uses a partition coarser than originally given. The second uses an additional "trap" state. The last one redirects certain transitions. Based on these results, in the next chapter, we present detailed abstraction and observabilization methods for EFSM represented by SDL specifications.

Chapter 4

Abstraction of States of SDL Processes

The EFSM concept is an underlying model for several languages and techniques for specification and development of protocols, interactive software, and hardware. These languages combine the EFSM model with complex structures of program languages, as procedures, control operators, and data types. Such languages could give a precise, effective, convenient, and familiar for the designer syntax (and, in the ideal case, a precise semantics) to specify communicating processes. The syntax of these languages is close to the syntax of common programming languages. Moreover, often, these specification languages are not just a "syntactic sugar" for EFSM, but considerable extensions of the basic EFSM model, which comprise many new complex, but useful concepts. Among these languages are Chill, Estelle, Promella, Lotos, UML, and SDL. Here we study adaptation of state abstraction and observabilization techniques developed in the previous section for the FSM and EFSM models for one of the above languages, Specification and Description Language (SDL). As discussed in Chapter 2, SDL has evolved within the telecommunication industry, however today SDL is also used in the development of other safety-critical systems, such as factory automation systems, aerospace and automotive applications, kidney-dialysis devices and train-control systems. We extend our methods for EFSM state abstraction and observabilization to SDL processes.

4.1 Correspondence between EFSM and SDL processes

It becomes clear from the Section 2.3 that SDL processes have several structures that are absent in EFSM. In fact, syntax and semantic discrepancies between the EFSM and SDL process create a gap between the two notations. This questions the applicability of the abstraction techniques developed for the EFSM model to simplify an SDL process. We try to identify this gap by establishing the correspondence between elements of the EFSM and SDL process and to bridge the gap to eventually develop an appropriate abstraction technique. We analyze how these discrepancies affect state and transition merging. At the same time, we indicate some implementation choices of our tool.

The EFSM is a theoretical model, but it gives a practical idea of hiding of less important or purely computational aspects into context variables and signal parameters, while representing more important control aspects with explicit states and signals. This idea is exploited in EFSM-based languages. However, the EFSM represents computations with variables and parameters in a quite abstract way – with predicates, context update function, and output parameter function. In the EFSM definition, it is not mentioned how these functions are represented, computed, or are they constructive at all. The variables types are not detailed at all either.

Below we list major discrepancies between SDL processes and our EFSM model, discuss their affect on consistency of our approach to state abstraction and observabilization. In this work, we do not offer a solution to all problems, moreover, not all proposed solutions are implemented in our experimental tool. However, we believe that identification of these problems clarify the way SDL states could be abstracted. Moreover, it could be helpful for further development of SDL abstraction and transformation tools and underlying methods.

Transitions. The correspondence between SDL transitions and transitions of EFSM is not direct. In SDL, transition usually means action statements which reside between states and are executed upon discrete or continuous signal arrival. Unlike EFSM, SDL transitions are branching on variable values (or non deterministically), i.e., they could lead into different states and invoke different action sequences. Starting state, triggers, terminators, should be considered as components of a transition. (However, the SDL syntax allows "transitions" which have no state or trigger, such as initial transition and transitions starting with label in-connector). Therefore, an SDL transition may correspond to a set of EFSM transitions. We suggest that a (semantic) state, an input, a chain of statements, connected in the graphical representation terminated with a *stop* or *nextstate* terminator, correspond to an EFSM transition. The predicates, updates and output functions are implicitly defined by that sequence of statements. Variable assignments in tasks and input parameters shape the context update function, and, often the output parameter function and the predicate guarding the transition. Decisions mainly contribute to the predicates. Expressions, used as actual parameters usually define only the output parameter function. The same statement may contribute to several different transitions of a corresponding EFSM. This complicates transition transformations, e.g., transitions merging. In some special cases, several chains leading to the same state could represent a single transition, using a non-deterministic choice between variable or output parameter assignments.

Outputs. An SDL transition may produce a sequence of output signals. If such a sequence could be determined by a static analysis of a process, it does not cause any problem for SDL transformation, for this sequence is considered as a single complex signal of a corresponding EFSM transition. Looping transitions do not complicate state merging however they complicate more complex transformation, i.e., observabilization. Our tool observabilizes only specifications with acyclic transitions.

Nondeterminism. While nondeterminism in EFSM arises when it has several transitions sharing the same input, standard SDL processes may have nondeterminism only in actions and decisions. These are ANY decision and ANY expression. Therefore, in the ITU SDL, nondeterministic transitions could be modeled only with ANY decisions. Fortunately, the OG extension of SDL allows nondeterminism in input clauses, hopefully this extension will be eventually accepted by the standardization committee.

ANY expressions are difficult to effectively represent in an EFSM model, but we do not see how this may affect neither state abstraction nor observabilization.

Time. Unlike our EFSM model, SDL processes are timed. Time, in fact, is considered as a designated variable, which may progress nondeterministically. Usually, in SDL processes, time aspects are expressed by a set of timers. Semantics of time progress and timers is complex and involves communication with implicit designated processes. Timers and time variables, used for timer setting, could be conservatively abstracted (removed), for example by replacing an input clause with a spontaneous transition (NONE input), though this is not implemented in the current experimental tool, because spontaneous transitions currently are not supported by the OG Simulator.

Procedures. Unlike EFSM, the SDL transition may call remote or local procedures. They complicate the variable dependency analysis. In general, the procedures may contain their own states and transitions. There exist methods and tools (such as SDL2IF, available from www-verimag.imag.fr) for flattening SDL processes, though their applicability in a general case is not clear. While a sound treatment of processes with procedures is possible, an auxiliary tool which we are going to use for variable removal in its current state does not support procedures. Therefore, we assume that all procedures (especially with states) have been flattered manually or automatically, otherwise variables involved in procedures will not be removed.

Discard. Inputs, which are not defined for a given state explicitly are discarded (consumed) by so-called implicit transitions. Therefore, to insure correctness of abstraction, implicit transitions should be defined explicitly. However, our tool does not automate it, all inputs should be manually explicitly defined for each state when a conservative abstraction is needed.

Queue (and related clauses). The SDL process has a queue (buffer) and a number of clauses to manipulate the order of consumption of the elements of this queue. There are explicit SAVEs, implicit saves (guarded transitions), and continuous signals. They do not breach the EFSM model, since a queue could be modeled by a variable (a dynamic array), saves could be modeled with transitions, which modify this array, continuous signals could be modeled with a designated empty input signal ε of EFSM, that triggers such a transition.

Moreover, SDL poses some syntax limitations, for example, it forbids the use of save and input for the same signal in the same state. Priority inputs can not appear in a state with a continuous signal. A straightforward merging of states with different continuous signals could make the state abstraction non conservative. (Here we view continuous signals different if their conditions or priorities are not identically equal. The continuous signal absence is equivalent to a continuous signal with an identically False condition. If continuous signals are the same, even followed by distinct transition, no problem arises). The problem is that a continuous signal clause of the merged state always fires, provided that no regular input transition is enabled and the condition of continuous signal is true. Therefore, in the absence of external signals, only the behavior of one merged state is reflected. A more sophisticated abstraction will provide a nondeterministic opportunity to stay in the state. A similar problem arises with priority inputs. Developing state abstraction for SDL processes, we will ignore all queue related operators, since it is not yet clear semantically how a conservative transformation can be achieved.

Shorthands. The SDL syntax exploits macros and a number of other shorthands, such as services, state lists, signal lists, input lists, star states and signals, dash nextstate, labels. These constructs simplify a process description, but may complicate state abstraction. Therefore, such shorthands have to be unfolded manually or automatically.

There are some other discrepancies between SDL and EFSM, we mention them to complete the list, but we do not see how they affect the SDL abstraction and observabilization.

Dynamic error (exception). It may happen, for example, in case of division by zero, when the list of answers for a decision is incomplete. Dynamic error can be also defined explicitly by the *error* keyword. In general, the behavior of a system after a dynamic error is undefined, though most of simulators can treat it. We will consider it as a special message.

Input parameters. Values of input parameters are assigned in SDL to variables. These variables contribute to the global state (configuration) space. Opposed to SDL, in EFSM, input parameters are among the arguments of the context update function. Parameters do not contribute to the configuration space. It is a technical detail, though it should be taken in account. An OG extension of SDL, namely implicit variables, allows one to use parameter values only in scope of a transition, without any contribution to the state space.

Termination. An SDL process can terminate. There are no designated terminators in EFSM. Termination could be modeled with a designated sink state of EFSM.

Alternatives and options are not considered. (Note that the latter are not supported neither by the OG editor nor by our tool.)

Extended communication schemes, such as via remote procedures, remote variables are not considered.

One of our means to bridge SDL and EFSM is the notion of an SDL machine, which is closer to EFSM than an SDL. We believe that most of SDL processes could easily be mapped into SDL machines. For complex manipulations with an SDL process (such as observabilization) we transform the SDL process into an SDL machine.

4.2 SDL State Abstraction

Consider state merging for SDL processes. We assume that a partition on states of a given SDL process is given. The partition defines the required abstraction of the original specification. The question is how states belonging to a single block of the partition could be merged into a single state of the resulting SDL process. Recall that states of the abstracted EFSM (see Section 3.3.1) were defined as sets of states of the original machine. Unlike the EFSM case, an SDL state name (identifier) can not be a set of other state names. SDL state names are alphanumeric strings. Therefore, we need to introduce a correspondence from state sets into state names. It is, of course, always possible to represent (finite) state name sets by state identifiers in an unique and unambiguous way using one-to-one mappings, but we prefer a less strict, but intuitive and clear method to use concatenations of state names. In the textual or graphical description of an SDL process, the same semantic state could be represented by multiple syntactical states. Therefore, to obtain the abstracted specification of the process, we just replace each entry of a state name by the concatenation of names of states from the given block in the partition. The order in which these state names are concatenated is not important, however it should be the same for each block. Therefore, state merging in SDL processes is performed by state renaming, where each state name in a process description is replaced with a new name, which is obtained by concatenating names of all states of the block.

The proposed transformation results in a syntactically correct (for OG version of SDL) process, which is a conservative abstraction of the original, provided that the original SDL process has no queue related operations, shortcuts, implicit transitions, and extended communication modes.

As we just stated, our method requires that an SDL specification be free of shortcuts (state or input list, asterisks, implicit transitions, etc.) We do not see any other way to correctly perform state abstraction of an SDL specification with shortcuts other than to first unfold these shortcuts.

State merging as proposed is not conservative for SDL processes that use queue dependent transition clauses, as it was explained above. The SDL syntax forbids the use of several instances of same input in same state (though, it is allowed by OG). Therefore, in the standard SDL, unlike EFSM model, a conservative state merging requires transition merging. Our tool is intended to work in conjunction with OG, but if the conformance to the ITU standards is needed, an ANY decision may be used in a merged transition to model nondeterminism. Figure 4.1 depicts from left to right a fragment of specification, a fragment of specification abstracted by a state name replacement (that what our tool produces), and the same fragment transformed to meet the ITU SDL standards.

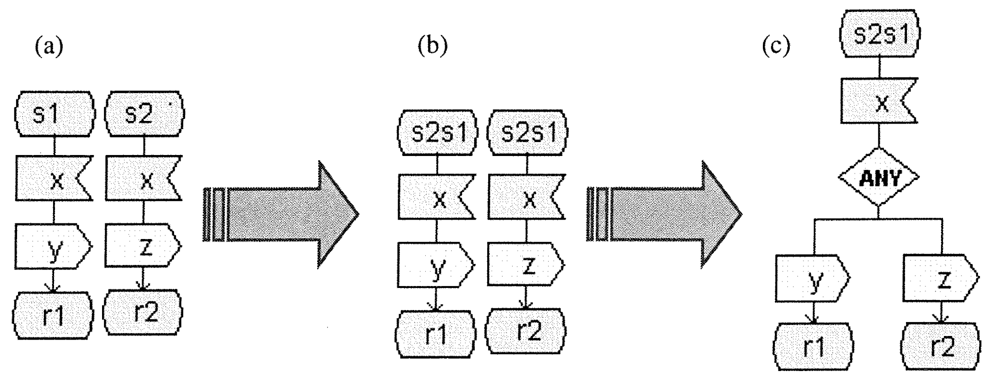

Figure 4.1 Merging States and Transitions

Now we discuss why and when merging of states with queue related operations by renaming could become incorrect or non-conservative, and suggest correct and conservative (w.r.t. the standard SDL) merging procedures. SAVES are considered first.

Having an input and a SAVE with the same signal for the same state is forbidden in SDL (in both, the standard and OG version). The SAVE of a signal is equivalent to the input guarded with an identical False enabling condition, we do not suggest any solution for explicit SAVEs, but concentrate on a more general case of implicit saves (enabling conditions). OG allows several possibly guarded input clauses for the same input signal and the same state. Though the OG tool provides a semantics in which if a signal is saved at least with one of these transitions it will be always saved, this does not completely preserve the intended behavior. Now, we discuss how a syntactically correct and conservative merging could be performed in the presence of implicit saves (enabling conditions, also known as guarding conditions). Since an input guarded with a (identical) False condition is equivalent to a save, only more general case of implicit save will be considered. A possible solution to add ANY (Boolean) expression into the enabling condition of merged and ANY decisions. Consider, for example, merging of two states, each with input x, guarded with conditions *i*>1 and *j*>2, shown in Figure 4.2, (a). The state diagram (b) presents the result of state renaming (as implemented in our tool). (c) diagram is a correct (w.r.t. standard) and conservative abstraction of these states. A refined diagram (d) of Figure 4.2 presents a finer, more precise, but at the same time, more complex abstraction of these states

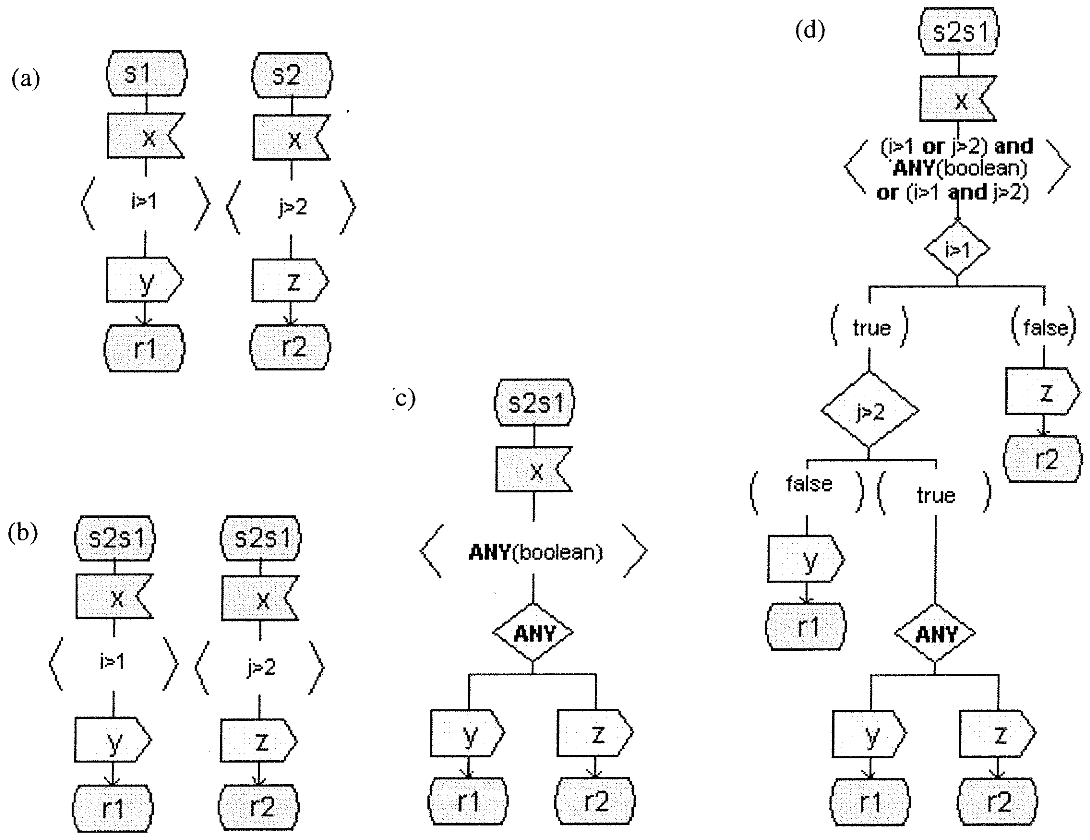

Figure 4.2 Merging States with Enabling Conditions

Note that a simple disjunction of original predicates do not lead us to a conservative abstraction.

The straightforward merging of states with continuous signals (or priority inputs) could be non conservative (as explained in the previous chapter). Tricks with ANY in the provided condition (as in the case of enabling conditions) could help here. Another problem is that it is forbidden by ITU standard [CCITT92] to have a priority input with a continuos signal it the same state. Generally speaking, a correct and conservative solution for the all above problems would be, instead of replacing old states with a new one, to add a new state with an asterisk save and True continues signal (or NONE) with an ANY decision or alternative which lead in these states (see Figure 4.3). Then, it may happen that some of states could still be merged. Such a solution does not always yield a specification simpler than the original one. In any case, it may be useful when the aim of abstraction is not just simplification, but also a strict preservation of the state behavior.

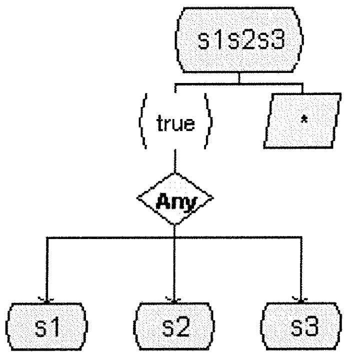

Figure 4.3 Abstraction by Introducing State

We have no general method capable of simplifying arbitrary complex SDL specifications, however, we have proposed a number of ad-hoc transformations, which are sufficient for simplification of "realistic" specifications.

4.3 Observability in SDL Processes

A specification may be either nonobservable due to or become nondeterministic as a result of abstraction. As it was explained in Chapter 3, an observable machine is more convenient than nonobservable, therefore, a tool for observabilization is required. For example, a method for test derivation presented in [PB+1999] requires observable EFSM specifications. Observable machines are easy to translate into observers (i.e., extended automata or acceptors).

By *observability* of an SDL process we understand the following property:
For each semantic state s, input x, and sequence of output signals $y_1...y_k$ there is a unique corresponding chain of connected statements from state s to a terminator nextstate or stop terminator. Statements are connected either directly in the SDL graphical representation of a process, or with a jump and a corresponding label.

We can derive the above notion from the EFSM observability notion if we assume that these chains correspond to EFSM transitions one to one. (i.e., the process is observable if a corresponding EFSM is observable). Though it is not the only possible way to establish a correspondence between processes and EFSM. Consider, for example, state diagram in the Figure 4.4.

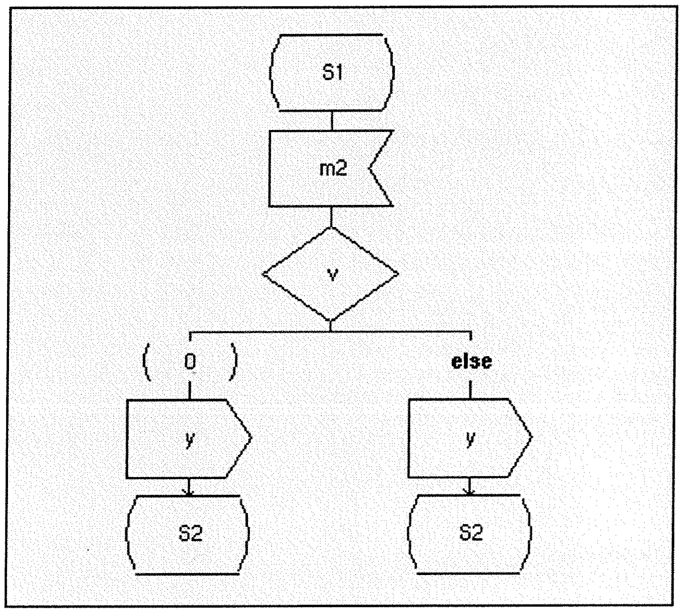

Figure 4.4 Branches with the Same Sequence of Output and Task Actions, Nextstate

We can model this fragment of a process either with two EFSM transitions

$$(s_1 - m2, v=0/y \rightarrow s_2)$$

$$(s_1 - m2, v \neq 0/y \rightarrow s_2)$$

or with a single transition

$$(s_1 - m2, (v=0) \lor (v \neq 0) / y \rightarrow s_2)$$

i.e., $(s_1 - m2/y \rightarrow s_2)$. So, if all transition paths of a state with same input, outputs, actions, output parameters, and next state are interpreted as a single EFSM transition, we obtain a weaker notion observability. We call it a weak observability.

An SDL process is called a *weakly* observable process if for every state, input and output sequence all the corresponding statement chains have the identical tasks sequence and output parameters.

Since weakly observable process may be (automatically) transformed into an observable one, we study first weak observabilization of SDL processes. It may be considered as a preliminary step of the observabilization.

4.4 SDL-machine

To simplify weak observabilization we assume that there are no joins and labels in the SDL process, and transitions of a state have a tree structure, i.e., there are no joins and labels, and decision branches never join again. Moreover, for the sake of simplicity, we prohibit timer and observation of global time, saves, implicit saves (guarded inputs), priority inputs, spontaneous transitions, joins, different shortcuts like state list, star symbols, input lists.

An SDL process satisfying this restriction is called an *SDL machine*. Consequently, to apply an SDL machine-based method for a regular SDL process we need to manually or semi-automatically transform the latter to the form close to the SDL machine.

Now we try to define an SDL machine structure in terms of components. Our definition concerns only the structure and we do not detail semantics and syntax of purely computational elements such as data types, expressions, procedures, however the latter are supposed to be free of states and signal passing.

SDL machine is a tuple of a finite set of parameterized signals, semantic states (state names), a variable set, a procedure set, a finite set of syntactical states, and start transition. The latter is used to define the first executable state and to initialize variables.

An SDL machine syntactic state is a set of transitions, which start from the same state, where an *SDL machine transition* is a tree, with a state and input clause in the root, nextstate or stop terminator on leaves, tasks, procedure calls, outputs, decision and answers as nodes. Decisions, except ANY decision, are followed with answer nodes. Answers are always placed after non-ANY decisions and only after them. The start transition follows above rules but it has no state neither input cause. Syntax states are introduced with the sole purpose to mimic SDL processes.

Input clause is a signal, optionally supplied with a list of variables, called parameters (not to be mixed up with input parameters of EFSM). The number of parameters and types are the same for all input clauses of a given signal, though several or all parameters could be omitted.

Output clause is a signal, which is supplied with an optional list of expressions, called output parameters. The number of parameters and type of each parameter are fixed for each signal, though all or several parameters could be omitted in this list.

Task is an assignment, procedure call, timer set or reset.

Decision is an expression or special ANY decision. It has at least two branches. The decision is the only node which can have more than one branch. If a decision is not ANY, it is followed by answers. *Answers* of a decision are expressions.

Functioning of SDL machines is not formalized. More work is needed to make the notion of an SDL machine more formal in the future. In this work, we assume that it coincides with a commonly accepted SDL process dynamics in this thesis.

4.5 Weak Observabilization of SDL Machine

Weak observabilization of an SDL machine can be performed with the following transformations.

1. For each state *s* and each pair of input and sequence of output signals labeling at least one path in the transition tree of the state *s*, find the set of all paths labeled with this pair.

   If the leaves of these paths are labeled with different nextstate values then label each nextstate with a designated "sink" state, where the "sink" state is defined as usually, i.e., for each input and output signal the sink state has a loop; output parameters are omitted.

   If the corresponding output signals of various paths differ in the values of output parameters, delete these parameters from the corresponding signals of all the paths.

   All variables, updated on the paths by assignments or procedure calls are placed in the list of "undesirable" variables.

2. Remove the undesirable variables along with all dependent variables and predicates defined by any of these variables.

Such transformation of an SDL machine will result in an observable machine, which preserves the behavior of the given process.

4.6 Conclusion

Identification of the discrepancies between an SDL process and the EFSM model can clarify the way SDL states could be abstracted and facilitates the adaptation of state abstraction and observabilization techniques developed for EFSM models to SDL processes. We have no general method capable of simplifying arbitrary complex SDL specifications, however, the concept of the weak observability of an SDL machine is sufficient for simplification of "realistic" specifications. In next chapter we present the algorithms and their implementation for state abstraction and weak observabilization.

Chapter 5

Experimental Tool Set

5.1 General Tool Set Description

We have presented several abstraction (simplification) techniques in Chapter 4, namely, for state abstraction and observabilization. To validate these techniques we implemented a set of experimental tools. Since our observabilization method is devised for SDL machines, any divergence between a real SDL specification and an SDL machine may cause semantic or syntactic problems. Therefore, to make process specification conformant with the SDL-machine notion, we also devise auxiliary programs for expansion of SDL transitions into a tree-like form, unfolding asterisks, and signal lists. State lists are unfolded automatically with the SDL API interface. No other transformation, i.e., elimination of saves, priority inputs, enabling conditions etc, is currently supported. If needed these transformations should be done manually.

The tool set developed here can simplify an original SDL specification and reduce the state (configuration) space. It may be useful for automatic test generation, verification and other related activities. For a restricted subset of SDL (see the constraints in pages 50-52) the abstraction is conservative in the sense that it preserves the behavior of the original processes of the SDL specification (though new traces may appear).

5.2 Caml – Our Choice for the Implementation Language

The programming language used for implementation is Objective Caml ver. 3.00 [Maun1995, Lero2000]. Caml is a state-of-the-art experimental functional language of the ML family. Caml is developed and supported by an effective free compiler mainly by efforts of INRIA. The compiler supports different environments and guaranties portability of the ML code. ML stands for a Meta Language and was developed mainly for purposes of program transformation and analysis, which perfectly suits our goals.

The family of strict functional programming languages has many advantages over conventional imperative languages. A program written in an imperative language consists mainly of statements. A statement takes its input values from the environment (such as variables), computes something, and stores the result again in the environment. To achieve a more complex behavior, we can execute one statement after another, or put statements into loops. Opposed to the classical imperative programming languages, functional programs are composed of functions. There are other benefits and advantages of functional programming:

1. There is no assignment statement. Thus, a programmer does not have to worry about storage allocation. Functional programmers usually do not assign values to storage locations. Instead, they give names to the values of expressions. These names could be used in other expressions, or passed as parameters to functions. For example *2*2* is a correct Caml program. *let x = 2*2* definition has the same sense as the phrase "Let's assume $x = 2 + 2$" in mathematics.

2. A name or an expression has a unique value that will never change. This is called referential transparency. Sub-computations will deliver the same result for the same arguments. It means that the code is safer and reusable in a similar context.

3. Functions and values are treated as mathematical objects, which obey well-established mathematical rules and are therefore well suited to formal reasoning. This provides the programmer a high level of abstraction and greater flexibility in defining control structures and data structures.

4. There are no restrictions in the definition and usage of functions, which can be passed as arguments or returned as values.

5. The rigorous mathematical basis provides functional languages with a clear semantics. The syntax and semantics of functional languages tend to be simple and so they are relatively easy to learn. Moreover, the programs tend to be more concise and have fewer mistakes.

Caml functions are based on the mathematically sound notion of a function, and thus differ from functions in conventional imperative programming languages. For example, there is no possibility to store something into the environment. The distinctive features of Caml are

1. Sophisticated data types such as lists, tuples, arrays etc., are predefined. A tuple is an ordered collection of data, for example *"x", true, 1* is a triple of string, Boolean, and integer. Type of a tuple is a Cartesian product of element types. In Caml, unlike Lisp, a list is a sequence of data of the same type. As a strict language, Caml provides only a limited support of infinite lists. Examples of list are: *[1; 3; 4; 5]* is a list of integers, *[]* is the empty list, *[[1; 2]; [1; 2; 3; 4]]* is a list of lists of integers. A standard library List facilitates operations with lists.

2. Caml offers powerful means to define new data types: records, enumerated types, and sum types. Records are Cartesian products (tuples) with names. For example, in our program the *bloc* type is a pair (2-tuple) of a string and a process list. The first element is labeled *bloc_name* and the second is labeled as *processes*:

```
type ...
bloc =
      {bloc_name: string;
       processes: process list}
```

Sum types can be thought as a generalization of union or variant types. They allow unrestricted definitions of heterogeneous values, tagged by data constructors. For example, in our program statements are represented by *stmt* data type, defined as sum (union) of *assign*, *output*, *decision*, *call*, and *terminator* data types, which are used for description of assign, output, decision, procedure call, or termination statements, respectively:

```
type ...
stmt =
   Assign of assign
 | Output of signal
 | Decision of decision
 | Call of call
 | Terminator of terminator
 | Label of label.
```

The word before the keyword ***of*** is the so-called constructor; the word after is a type.

3. Caml supports pattern matching for the sum (variant) type. Patterns are templates that allow selecting data types of a given shape, and binding identifiers to components of the data types. This selection operation is called pattern matching; its outcome is either "this value does not match this pattern", or "this value matches this pattern, resulting in the following binding of names to values". Pattern matching facilitates symbolic computations. The following is an example function, which takes a *stmt* data (representation of a statement) and returns the true Boolean value for a label and false otherwise.

```
let isit_label a_stmt =
  match a_stmt with
    Label l -> true
```

```
      | _ -> false
```

A slightly shorter version could be

```
function Label _ -> true | _ -> false
```

The following is a recursive definition of a function

```
let rec factorial = function 0 -> 1 | n -> n * factorial(n-1)
```

4. There is no need to add type information in programs (as in Pascal or C): type annotations are fully automatic.

5. Caml is a safe language. The compiler performs many sanity checks on programs before compilation. Allocation and deallocation of data structures is kept implicit (there is no "new", "malloc", or "free" primitives). A general exception mechanism supports error recovery.

6. The module system and separate compilation simply development of large projects.

7. Caml provides full imperative capabilities, when needed.

There are certain problems that complicate wide usage of functional languages in industry, as lack of competent programmers, libraries, and fancy GUIs, sometimes complexity of efficient implementation of some algorithms. Imperative programming even if supported could be cumbersome in functional languages. However, functional languages, and especially Caml, perfectly suit for rapid development of complex experimental/prototype CASE tools. For our purposes of producing an experimental tool a high level language as Caml is preferable.

5.3 Caml-SDL API and Abstract Syntax Tree

The implementation of the algorithms is based on the CAML-SDL API supplied by France Telecom R&D. This API allows one to parse a textual SDL specification and extract an abstract syntax tree (AST) and pretty prints a given abstract syntax tree into a textual (pr) SDL specification. Some transformation, such as unfolding state lists, transformation of timers into input signals are done automatically. API supports a sufficiently large subset of SDL. Since all transformations are performed on AST of SDL specifications, we describe how exactly this tree is defined. The data types are divided into two classes. The first class corresponds to the static structure of the SDL-specification, which are data types *spec* and *bloc*. The second class mainly corresponds to the dynamic structure of the SDL-specification, that is the part of EFSM model. The data types of this class include SDL components, such as process, procedure, state, clause, input, signal, statement, and some other related data types. The data types are organized in a tree-like structure. The root of the tree is data type *spec*. Each node of the tree is a data type. The up level structure provides some fields to access the next level of the SDL specification. Figure 5.1 shows the data types and their dependencies defined in the CAML-SDL interface.

From Figure 5.1, we can notice that the data types above stmt list are a tree-like structure and the data type stmt is recursive. Some statements, namely d*ecisions*, contain lists of *stmt* (statements). Each statement list is associated with a branch of a decision. This is a widely accepted representation of a tree in functional programming. An SDL process is represented as a tuple of

A process name,
Input messages,
A variable list,
A state list,
A procedure list.

```
process =
     {process_name: string;
      messages_in: string list;
      variables: Var.variable list;
.....(* some elements are skipped *)
      typ_variables: Var.types_de_variables;
      states: state list;
      procedures: procedure list}
```

According to this kind of structure, in our implementation, typically, we first access each node of the spec using the tree traversal algorithm.

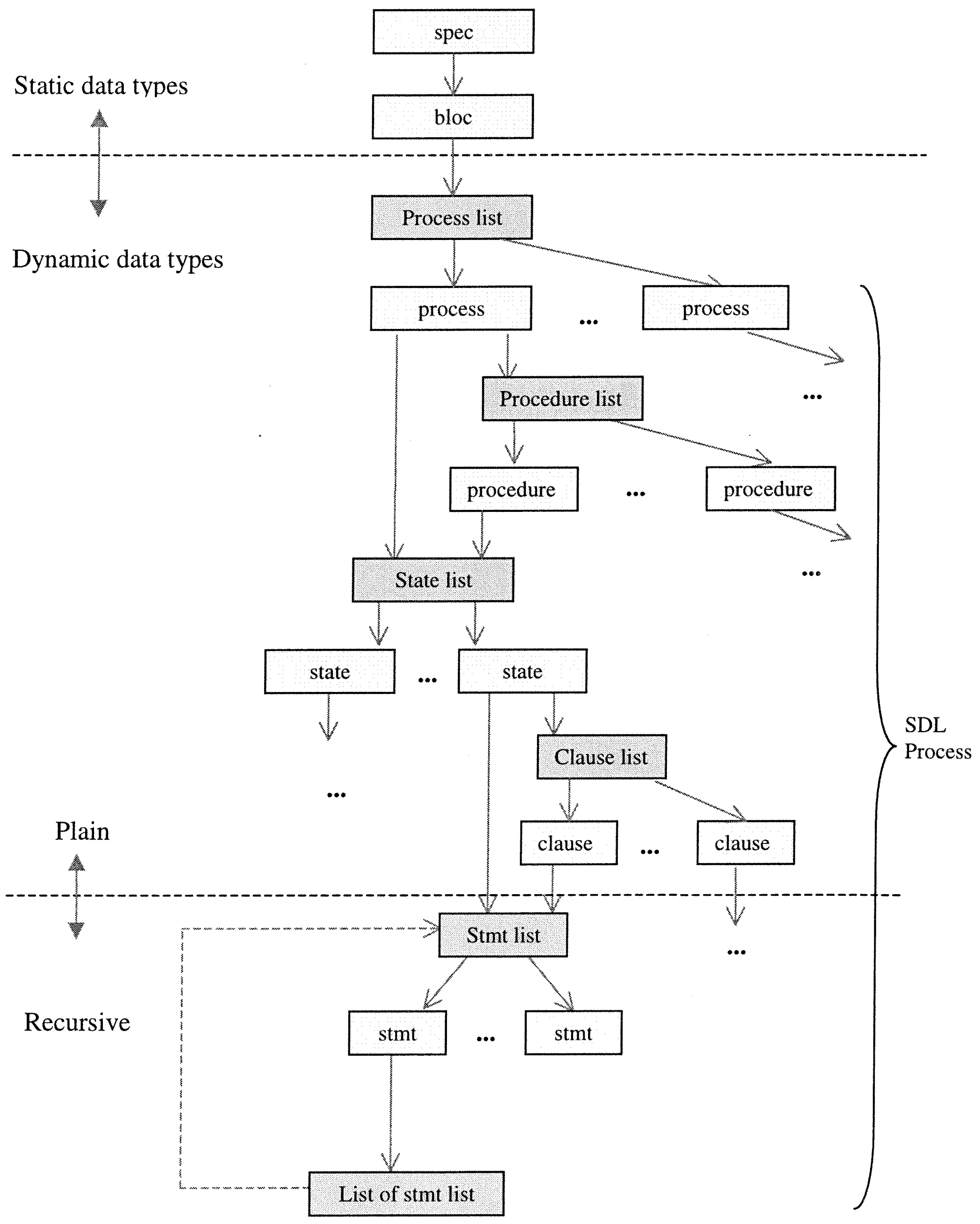

Figure 5.1 CAML-SDL Interface

5.4 Patterns and shapes of the SDL Abstract Syntax Tree

In this section we describe other data types of SDL API Abstract Syntax Tree (AST). Most of them are variant types. The variant (a.k.a. sum) type is declared by listing all possible shapes (or patterns) for values of the type.

State

Data type *state* represent SDL (syntax) states and starts, therefore it comes in two shapes:

```
state =
      Start of stmt list      (* list of statements = body of a
                              start transition*)
   |  Normal of (string * clause list) (*Pair of a state name
                                        and list of clauses*)
```

The first shape is for start. It follows the statement list that describes the start transition. The second shape allows representing of a syntactic state of the process. It consists of a state name and a clause list. Clauses represent outgoing transitions of a state.

Clause

A clause has five shapes:

Save,

Save_star,

Input,

Input_star,

Cont_sig.

```
clause =
    Save of string list
  | Save_star
  | Input of input_cl
  | Input_star of input_star_cl
  | Cont_sig of cont_sig_cl
```

The first two are for the two shapes of "Save" clause. The next two are for the different shapes of "Input".

```
input_cl =
    {signals: signal list;
     garde_opt: Var.expression option;
     stmts: stmt list;
     parametres_non_signifiants: bool} and signal =
    {sig_name: string;
     sig_params: (Var.expression * Var.un_type) list}
and input_star_cl =
    {garde_opt_star: Var.expression option;
     stmts_star: stmt list}
```

The continuous signal clause type is defined as a pair of enabling (guarding) condition and a statement list (transition body).

```
cont_sig_cl =
    {garde: Var.expression;
     stmts_cont_sig: stmt list}
```

Statement

In SDL, there are six main type statements, TASK, OUTPUT, DECISION, CALL, TERMINATOR and LABEL. The date type for statement *stmt* could be in one of the following six shapes:

Assign

Output

Decision

Call

Terminator

Label

```
stmt =
    Assign of assign
  | Output of signal
  | Decision of decision
  | Call of call
  | Terminator of terminator
  | Label of label
```

Terminator

Data type terminator has three shapes:

Return,

NextState

Join .

```
terminator =
  Return | NextState of nextstate | Join of label
```

Nextstate

There are two shapes for nextstate:

Minus or

NoMinus of label.

```
nextstate =
    Minus | NoMinus of label
and label = string
```

Decision

SDL API represents an SDL decision with a type of the same name. It consists of expression, which describe a condition, list of regular branches, and a possibly empty "else" branch. Each regular branch is a pair of answer (*expression* type) and a transition body (*stmt* list).

```
and decision =
    {var_testee: Var.expression;
     branches: (Var.expression * stmt list) list;
     branche_else: stmt list}
```

Decision is a recursive type; an instance of *Decision* could contain other decisions in some branches.

Other types are obvious from the context.

5.5 Underlying Algorithms

Since ML is a formal language, supported by numerous tools, and many experts on functional programming claim that a program on formal functional language, as ML, is a self-contained and complete description of the underlying algorithm, and a detailed informal description in English, state charts, etc may only mislead the reader. However, for those, who do not share this opinion, or are unfamiliar with functional programming, we represent an informal (approximate) description of the algorithms.

All programs of the tool set take as the input and result in a textual SDL specification. However, our algorithms are only concerned with a process transformation and performed on the Abstract Syntax Tree.

5.5.1 State Abstraction Algorithm

The EFSM state abstraction goes as follows. Let A be a completely defined EFSM with the set of states S and $\pi = \{B_1, B_2, \ldots\}$ be a partition on S. According to the definition of the factor machine in Chapter 3, we know that a factor for A is an EFSM whose each state is a block of the partition π, the initial state of EFSM belongs to initial state of the factor EFSM, and, for each transition $(s—P, x/y, up, op\rightarrow s')$ of A, there exists a transition $(\pi(s)—P, x/op, y, up\rightarrow\pi(s'))$ and vice versa.

As was described in the previous chapter, the algorithm for state abstraction performs coping of the process while replacing each state name with a new name, designating to the corresponding partition block of this state.

Algorithm 1. The state abstraction in a process.
Input. AST of a process, list of non-singleton blocks of a partition on (semantic) states.
Output. An AST of the abstracted process

1. Build a new syntax tree of a process as follows.

2. Keep the same name of the process and the same data declaration part.
3. Define a new start (called here for simplicity start state) from the given start state as follows.

   Each element of the new start transition coincides with the original, except that in nextstate statements state names are replaced with names of their partition block.
4. For each syntactic state of the original process (i.e., the description of a set of transition between semantic states),
   - 4.1 Define a new syntactic state of the abstracted process, where instead of the original name of the state the name of the corresponding block is used.
   - 4.2 For each input clause/save/continuous signal of the original syntactic state define the same input clause/save/continuous signal. Define corresponding transition bodies in the same way as in the original, except for nextstate statement, where each state name is replaced with the name of the partition block.

Remark 1. Here the name of a partition block is a concatenation of names of all states which forms this block.

Remark 2. Caml program does not require to specify a process, it just chooses the first process in the block.

The following is the Caml implementation of the algorithm. Details of the main procedure, which are irrelevant to the algorithm, are briefly described in English.

(*main function, module state_merge_process *)
(*takes the name of a pr textual SDL specification from the command string argument, loads AST of this specification with the SDL API, enters the partition of states of the first process, computes the tree of abstracted specification with the function process_merge of the process_merge module, and pretty prints it *)

```
let main () =
        Get the name of a pr file from the argument list (fname);
        Parse this specification with an SDL API and get an AST spec0;
        Print the list of states and ask user to enter the non-singleton blocks of the
        partitions one by one, as well as the number of blocks;
        Input the number of non-singleton block block_number, then blocks
        themselves, and store them as array block_list of string lists, where each string
        is a state name;
        Form an array of names of block merge, where each name is concatenation of
        names of states in the block;
        Pretty print an AST computed as process_merge spec0 block_list merge
        block_number into the textual specification with a name fname with a suffix
        merge.
```

5.5.2 Algorithms for Transforming SDL Transitions into a Tree Form

Decision Flattening

A decision statement consists of a conditional expression, a non-empty list of normal branches and an ‘else’ branch. A normal branch consists of an answer (which is an expression) and a statement list. The ‘else’ branch is a possibly empty list of statements. A decision statement can be recursive. The special case for decision is that, following the branches, it may have a set of common statements as shown in Figure 5.2(a). For the sake of the observabilization, this kind of structure needs to be flattened; that is, we have to attach these statements to each branch of the decision as shown in Figure 5.2(b).

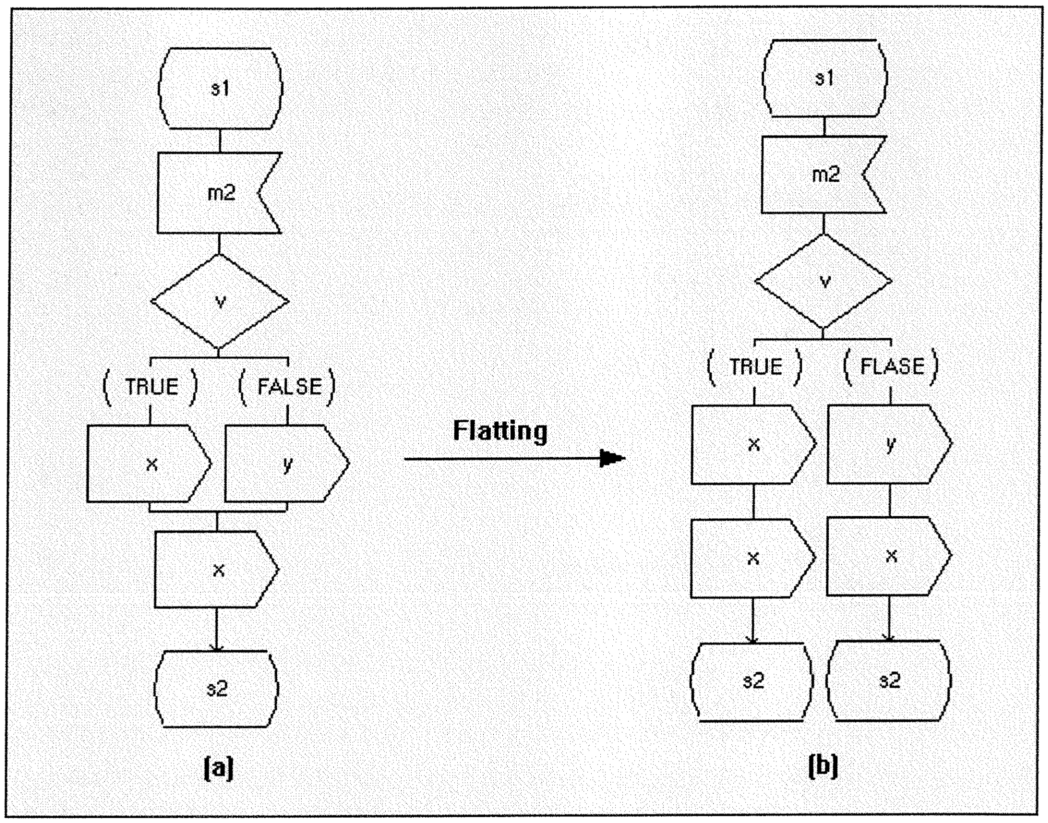

Figure 5.2 Flattening the Decision

The flattened AST of a process is a tree of a process with a state set built as follows.

Algorithm 2. Unfolding of merging decision branches.

Input. A process AST

Output. An AST of a process with flattened decision branches.

1. Define a new start with a stmt list computed from the stmt list of the original start.
2. For each state of the original process define a state of the new process with the same name, saves, input clauses, continuous signal with transition bodies (stmt lists) transformed by the algorithm 2.1.

Algorithm 2.1. Unfolding branches in a stmt list (an auxiliary algorithm for the algorithm 2)

Input. stmt list. (List of statement – assignments, outputs, terminators, decisions with all branches included, or a labels)

Output. A semantically equivalent stmt which does not contain merging branches of decisions.

1. Find the first element of the list *fst* and the rest *tail*.
2. If the first element of the list *fst* is not a decision, keep the first element without changes and apply this algorithm to the *tail* recursively.
3. If the first element is a decision, the result is the list that consists of a single decision stmt. Build this decision as follows from the *fst*

   3.1 apply this algorithm to the list of stmts of each branch

   3.2 attach *tail* to each list.

Label Replacement

Here we describe an algorithm for eliminating acyclic joins (out-connectors) and detecting cyclic joins. Each acyclic out-connector is replaced with a corresponding fragment of a transition. We need to scan the given specification twice. In the first scan, we construct a pair list. The first element of the pair is a label name; the second is a statement list that follows the label. With the second scan we build an equivalent process which has no connectors (joins). The joins are iteratively substituted with the corresponding transition fragments.

Algorithm 3. Unfolding connectors.

Input. A process.

Output. An equivalent process without joins (out connectors).

1. Build the list L of pairs of label and stmt list by applying Algorithm 3.1
2. Build a process by applying Algorithm 3.2 to each transition.

Algorithm 3.1 Label list

Input. An SDL process AST

Output. List of labels along with corresponding stmt lists.

1. Find the list of stmt making transitions of the process.
2. Apply Algorithm 3.1.1 to each stmt list.
3. Merge the results into a single list. Stop.

Algorithm 3.1.1 Extraction of labels with stmt list from a stmt list.

Input. Stmt list

Output. List of labels with stmt list

1. If stmt list is empty then the result is the empty list.
2. Otherwise, i.e., if the list consists of a head and a tail do
3. If the head is a label then the result is the list with the first element pair (head, tail) and the rest is computed by applying this algorithm to the tail recursively.

Algorithm 3.2. Unfolding connectors in a stmt list (a transition body)

Input. Stmt list, list of labels along with corresponding stmt lists.

Output. Connector free stmt list.

1. If the stmt list is empty the result is the empty list.
2. If the first element of the stmt list is a Join j, then
   - 2.1 Find the list of stmt corresponding to the label j
   - 2.2 Apply this algorithm to it, the resulting list is the result
3. If the first element is a decision d.
   - 3.1 Denote the rest of the list as tail
   - 3.2 Build a new decision by applying this algorithm to all the branches of this decision.
   - 3.3 Apply this algorithm to the tail, attach new tail to the decision. This is the result
4. If the first element is a label, then compute the result by applying this algorithm to the tail. (label is discarded).
5. Otherwise, the first element of the result is the same, the remaining part is computed by applying this algorithm to the tail.

5.5.3 Algorithm of Observabilization

The state merging of an EFSM can result in nonobservable transitions and repeated transitions. The observabilization algorithm takes care of these problems. Before we give the details of this algorithm, we present a definition that is used in this algorithm.

Definition 5.1 State-Input-Output-NextState Sequence (SION-Sequence) of an EFSM:

Given a transition (s—P, $x/y_1y_2...y_k$, up, $op \rightarrow s'$) of an EFSM, a sequence of the source state s, input x, outputs y_1, y_2, ... , y_k, and the next state s' is called SION Sequence. E_{SION} is the set of all state-input-output-nextstate sequences of the given EFSM.

Here we aim only at weak observabilisation, i.e., transformation that ensures that transitions of the same state, input and outputs could differ only in predicates (decision conditions), see Section 4.5. Observabilization consists in the following transformations.

1. For each state s and each pair of input and sequence of output signals labeling at least one path in the transition tree of the state s, find the set of all paths labeled with this pair.

   If the leaves of these paths are labeled with different nextstate values then label each nextstate with a designated "sink" state, where the "sink" state is defined as usually, i.e., for each input and output signal the sink state has a loop; output parameters are omitted.

   If the corresponding output signals of various paths differ in the values of output parameters, delete these parameters from the corresponding signals of all the paths.

All variables, updated on the paths by assignments or procedure calls are placed in the list of "undesirable" variables.

2. Remove the undesirable variables along with all dependent variables and predicates defined by any of these variables.

We provide algorithm only for the first step the second step could be implemented with another tool, written by Xiaoyu Wang [Wang2001].

Algorithm 4. Weak observabilisation

Input. A process.

Output. A process and list of variables to delete.

Phase 1:

Find state-input-output-sequence strings in the non-observable transition path:

1. Construct SION string sequences for all transition paths in the given spec.
2. Determine and report the variables need to be deleted.
3. Determine all SION-sequences in non-observable transition paths:

   For those with same nextstates: Keep the next state name unchanged.

   For those with different nextstates: Convert the SION sequence into an array and then put this array to an array list. In this array, the first element is a state name; the second element is an input signal name; the rest elements are the output signal names in this path attached to the state and input.

Phase 2:

Redirect to the “Sink” state.

For each array in the array list of Phase 2 do:

For each state of the original process

if the state name is equal to the first element of the array, reconstruct its clause list such that each signal list in an Input clause only contains a single signal.

For each clause in the new formed clause list, process the following statement list to see if the rest elements of the array are the same as the output signals in this path. If yes, replace the nextstate name with name “Sink”.

Phase 3:

Add the new state "Sink" to the state list of a process.

For each block in spec do:

For each process in a block do:

1. Add state "Sink" into the state list
2. Add a new clause list to this "Sink" state, that is, create its self-looping (trapping) transitions with all possible input-output labels.

5.6 Details of the Tool Set Implementation

This tool set uses the Object GEODE SDL-92 Application Programming Interface (SDL API) [GEODE-3] to load an SDL specification textual file into a C data structure. An interface, provided by France Telecom R&D, transforms this C data structure into a ML data structure and allows printing out of this data into an SDL-like form. The latter is not a valid SDL; however, it can easily be transformed manually into a valid SDL by the user by adding the signal and route/channel description.

For an easier implementation and maintenance a module design strategy is used in the implementation. The implementation of the tool set consists of four modules, state_merge_process, decision_process, label_process and weak_observ_process. The first one performs merging of the states of a process according to a specified partition, and the second and third perform transformation (unfolding) of the process transitions in a form convenient for observabilization. The fourth program, weak_oberve_process, observabilizes a process. Figure 5.3 describes the modules of the tool set and the dependencies between the modules.

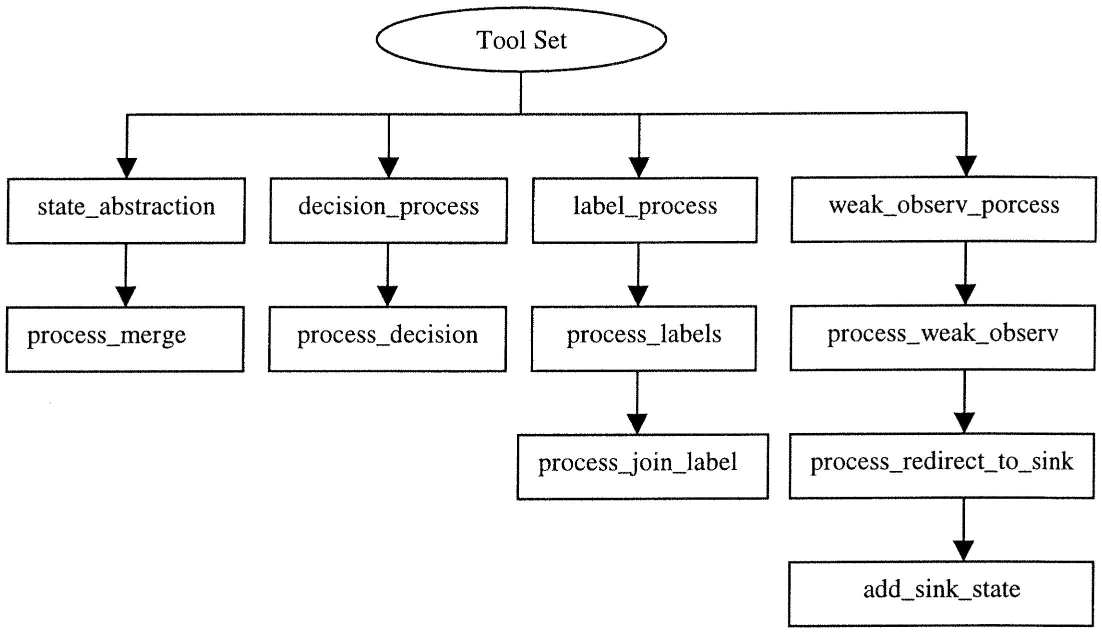

Figure 5.3 Modules of the Tool Set

5.6.1 State Abstraction Tool

State abstraction tool "state_merge_process" is designed to merge states that belong to a single block of the partition described in section 5.2. This tool preserves behavior of the original machine, while possibly adding a new behavior. It gets the SDL-specification file from the command line argument and uses the Object GEODE SDL-92 Application Programming Interface (SDL API) and the ML SDL API to load the SDL specification textual file.

The *state_merge_process* receives the specification name as a command line argument. Then, the program outputs names of states of a process, and the user defines constructs the specified blocks of the partition interactively and then use the submodule *process_merge* to merge the states in each block (Note, the block here is a set of states. It is different from the "block" concept in SDL). The program performs state merging.

The submodule “process_merge” merges the states in each block of a partition. It returns a type “spec” with all information changed. The module process_decisions takes this “spec” as its parameter and unfold decisions with merging branches. The submodule process_decisions consists of seven functions. The submodule is designed to scan the SDL system from the top-level spec and trace the each transition path to the leaf-level terminator. The result is written into a file. The name of the file consists of the name of the original specification file and a suffix *merge*.

The functions of this module are

– process_merge
It is the entry of the submodule (the function of the module used in the main function). It calls function process_in_bloc to enter the next level node, that is block and returns the result of the bloc type.

– process_in_bloc
It applies the function “process_in_process” to each element of the process list in the block one by one and results in an SDL block with a new process list.

– process_in_process
It has two functionalities: First, it applies the function “rename_in_state” to each element of state list in a process and forms a new state list of this process. Second, it applies “process_in_procedure” to each procedure in the procedure list of the process and forms a new procedure list.

– process_in_procedure
This function is used to treat states of a procedure by applying the “rename_in_state” function to each element of state list in the procedure and forms a new state list of this procedure.

– rename_in_state

If the argument is a start this function applies function "rename_in_stmt" to each element of the stmt list and results in a start with a new stmt list. If the argument comes in the form "Normal (state_name, clause_list)", it replaces the name of the state with a new merged name and then applies "rename_in_clause" to each element of the clause_list. The result is a state with a new clause list.

– rename_in_clause

This function is used to treat different shapes of clauses. It applies "rename_in_stmt" to each element of the corresponding statement list.

– rename_in_stmt

This function is used to treat a terminator statement. If the argument shape is "NextState of nextstate" and the name of nextstate is in the set of merged states, it replaces the name with the merged name. It does not make any change to other shapes.

5.6.2 Tools for Transformation of SDL Transitions into a Tree Form

The tools *decision_process* and *label_process* transform the transitions into a tree like form, which is convenient for observabilization. It unfolds merging branches and eliminates out-connectors. Each out-connector is replaced with a corresponding (sub) transition. However, the connectors should not be cycled, otherwise the program does not stop.

- **Module process_decisions**

Submodule process_decisions takes a "spec" (an AST of an SDL spec) as its argument to unfold decisions with merging branches. The submodule process_decisions returns a new "spec" with all changed fields. There are seven functions in the submodule.

– proces_decisions

It is the entry of the submodule. It calls function process_decision_in_bloc to enter the next level node, that is block, and return a specification with the new block.

– process_decision_in_bloc

It applies "process_decision_in_process" to each element of the process list in the block one by one and returns a new process list of the block.

– process_decision_in_process

First, it applies the function "process_decision_in_state" to each element of state list in a process to form a new state list of this process. Second, it applies "process_decision_in_procedure" to each procedure in the procedure list of the process to form a new procedure list.

– process_decision_in_procedure

This function is used to treat states in a procedure by applying function "process_decision_in_state" to each element of the state list in the procedure to produce a new state list of this procedure.

– process_decision_in_state

For a Start state, it applies the function "process_decision_in_stmt" to each element of the its list "stmt_list" to trace the corresponding outgoing transition path from "Start" state to its terminating state and results in a new statement list. For a Normal state, it applies the function "process_decision_in_clause" to each element of the clause_list to trace each of its outgoing transition paths to the terminating state in each path. It results in a new clause list.

– process_decision_in_clause

This function is used to process different shapes of clauses. It applies "process_decision_in_stmt" to each element of the corresponding statement list.

– process_decision_in_stmt

This function processes each Decision statement recursively to see if there is a merging statement for this Decision statement. If it is, it calls the function process_decision_merge.

– process_decision_merge

This function is used to append a stmt list to each branch of the Decision statement.

- **Module label_process**

The module label_process is responsible for unfolding connectors (joins) in the process transitions. It uses module process_labels to finding out all labels and the statement list of each label and then uses submodule process_join_label to unfold joins and delete all labels in this new "spec".

Submodule process_label

This submodule consists of six functions. Each function is described as follows.

– process_label

This function is the entry of the submodule. It applies function process_label_in_bloc to the specification block

– process_label_in_bloc

This function applies function process_label_in_process to each process of the block.

– process_label_in_process

This function processes each Label statement in the process of the given specification. It first constructs a label list of this process by calling function process_label_in_state. This list contains all the labels in the process. Each element of the label list is a pair (label name, tail-statement list). The tail statement list is a list followed this label. Then it splits this list of pairs into a pair of lists. Finally, it calls the submodule process_join_label to process the Join statement in the process by means of the formed lists. After processing each process, the functions "Utils.log" and "Spec_sdl.pretty" are called to output the changed SDL specification to a new PR file.

– process_label_in_state
For a Start state, it applies the function "process_label_in_stmt" to each element of the its list "stmt_list" to trace the label in outgoing transition path from "Start" state to its terminating state and results in a new statement list. For a Normal state, it applies the function "process_label_in_clause" to each element of the clause_list to trace the labels on the transition paths outgoing from this Normal state. It results in a new clause list.

– process_label_in_clause
This function is used to process different shapes of clauses. It applies "process_label_in_stmt" to each element of the corresponding statement list.

– process_label_in_stmt
This function processes each Label statement. If it meets a Label statement, it calls function loop_detect to see whether or not there is a loop that exists in its tail list. If it is, it processes its tail list using process_label_in_stmt recursively. If it is not, it outputs a (label name, tail list) pair to a list and process the statements in the tail list recursively. If it meets a Decision statement, it processes the statement in each of its branches recursively.

– loop_detect
This function detects the loops that exist in the transition path and reports them.

Submodule process_join_label

The submodule process_join_label eliminates a connector found in the submodule process_labels and removes all labels in the "spec". This submodule carries four parameters, spec, a label name that needs to be removed, the tail list followed this label, and a pair of list. Eight functions are included in this submodule.

– process_join_label
This function is the entry of the submodule. It applies the function process_join_label_in_process to block.

– process_join_label_in_bloc
It applies "process_join_label_in_process" to each element of process list in the block one by one and results in a new process list of the block.

– process_join_label_in_process
This function applies the function "process_join_label_in_state" to each element of the state list in a process and forms a new state list of this process.

– process_join_label_in_state
This function is used to process different shapes of states. For a Start state, it applies the function "process_join_label_in_stmt" to each element of the list "stmt_list" to trace the join labels in an outgoing transition path from "Start" state to its terminating state and results in a new statement list. For a Normal state, it applies the function "process_join_label_in_clause" to each element of the clause_list to trace the join labels on the transition paths outgoing from this Normal state. It results in a new clause list.

– process_join_label_in_clause

This function is used to treat different shapes of clauses. It applies "process_join_label_in_stmt" to each element of the corresponding statement list of a clause.

– process_join_label_in_stmt

This function is used to remove the Label statement and Terminator from a statement list. When a Terminator statement is the head element of the argument, the function first checks if it is in the form of Join of label, if it is, it discards this statement and processes its tail list recursively. When a Decision statement is the head, the function processes statements in each branch recursively. When a Label statement is the head and the label name in this statement is same as the label name carried by the submodule process_join_label, the function discards this label and processes its tail list using the function process_tail_list. If the label name is not the label that it intends to remove, the function outputs this statement to the statement list and processes its tail list recursively. For other cases, the function outputs the statement into statement list and processes the tail list recursively.

– process_tail_list

This function is used to remove the labels in the tail list.

5.6.3 Observabilization Tool

The *weak_observ_process* receives the name of the specification from the command, and then it transforms a process of the SDL specification into a weak observable form. Namely, the program outputs a list of variables, which should be deleted to obtain a weak observable specification, and redirects some transitions into a sink state. The result is written into a file. The name of the file consists of the name of the original specification file and a suffix *weak.*

The removal of variables may be performed with the existing tool for abstraction of variables. However, the latter tool could be applied only to a valid SDL specification. Therefore, there are two possible strategies for obtaining an observable specification. If the user needs the valid SDL specification, he first performs state observabilization with the *weak_observ_process* program, stores the list of variables that should be removed, then manually debugs the resulting specification, and finally applies the tool for variable abstraction. If the valid SDL specification is not needed, the user could use the program *weak_observ_process* to derive a list of variables to delete, then apply the tool of variable abstraction to the initial specification, and, finally, apply the *weak_observ_process* to the result to redirect required transitions into the sink state.

There are three submodules in this module. The dependence between the functions is shown in Figure 5.3. The module *weak_observ_process* calls the submodule *process_weak_observ* with the "spec" of the SDL-based specification resulted. The submodule *process_weak_observ* calls *process_redirect* to redirect the nonobservable transitions to into the sink state. The submodule *add_sink_state* is called for adding a new state "Sink" into the state list of a process. The following is the description of the implementation of the weak observabilization tool.

Submodule process_weak_observ

The submodule sink_process can parse and perform syntactical and semantic check of an SDL specification, find out the state-input-output sequences, check the repeated sequences and their corresponding transition path, report the variable that need to be deleted and then redirect the nonobservable transition paths to a "sink" state. It uses *process_redirect* and *add_sink_state* modules and is composed of eleven functions.

– process_weak_observ

This function does almost the same as the function "process_state_merges", it also carries "spec" as its input parameter, but instead of returning a type "spec" with all changed fields, it changes only the necessary fields of spec. At the end, it calls "Utils.log" and "Spec_sdl.pretty" functions provided by France Telecom R&D to output the changed SDL spec into a new PR file.

– process_in_block

This function is used to treat a block in the system.

– process_clause_in_process

This function applies the function "treat_in_state" to each element of the state list of a process. Treat_in_state produces a list, which contains the SION sequence in nonobservable transition paths for each state. The function uses the submodule process_redirect to redirect the nonobservable transition path to a sink state using this list as its parameter. After handling all elements of the list, the function uses submodule add_sink_state to add a state "Sink" to the resulted specification file.

– find_variables

This function is used to return the variable name in the left side of a Task statement.

– treat_in_state

This function has several functionalities. First, it reports all transition paths from a state. Second, it constructs a SION-sequence string list for each state. Each sequence in the list corresponds to a transition path. It returns this list to the function "process_clause_in_process" and the function "process_clause_in_process" uses this list as its parameter to call the submodule process_redirect. Finally, it detects the non-observable transition paths, finds out the variables need to be deleted, and reports the corresponding state-input-output sequence.

– treat_in_clause

This function processes each clause. First it applies function "treat_signal" to each element of the signal list of the clause and then returns the results in a list. Each element of the list is a list of state_input_output_nextstate_name sequence string list.

– treat_signal

This function is used to append an input signal name to the beginning of the statement list.

– treat_in_stmts

This function processes a statement list in each path recursively to construct a state_input_output_nextstate_name sequence string list.

– process_clause_in_path

This function processes each SDL clause. It applies the function "process_signal_in_path" to each input signal in the input signal list to form a list of list of statement list.

– process_signal_in_path

As we discussed in Chapter 4, in the input symbol of SDL, instead of a single consumed signal, there can be an input signal list. It is possible that some input signals can cause nonobservable paths, while others can not. To treat such a situation, we have to process each input signal individually. This function processes each input signal and passes the statement list to the process_stmt_in_path.

– process_stmt_in_path

This function processes SDL statements in each path recursively to construct a list of statements in a path; the last statement in the list is "NextState" statement.

Submodule process_redirect_to_sink

This function is used to redirect a non-observable transition with the same next state to a "Sink" state. It carries a string array list as its parameter. It includes the following functions:

– process_redirect_to_sink

This function is the entry of the submodule.

– process_clause_in_bloc

This function applies function process_clause_in_process to each process of process list in the block.

– process_clause_in_process

It applies the function redirect_in_state to process each state in the list and applies the function process_clause_in_procedure to process procedure list in a process.

– process_clause_in_procedure

It applies the function redirect_in_state to process each state in the state list of a process.

– redirect_in_state

This function is used to process different shapes of states. For a Start state, the function outputs it directly to the generated specification file. For a Normal state, if this state is the state in the list, there are two things to be done. It applies the function "redirect_in_clause" to each element of the clause_list.

– process_clause

This function is used to produce a new Clause list. In the case that a state has a signal list as its input and all of these signals have the same transition path and reach the same NextState, this function is used to flat the signal list by means of the function treat_signal.

– treat_signal

This function is used to form an Input clause with the input signal list that contains only a single input signal.

– redirect_in_clause

This function processes each clause of the new formed clause list. For each signal, which is in the SION list, it traces the statement list of this signal by calling the function redirect_in_stmt.

– redirect_in_stmt

It processes the statement list of a signal to see if all output names and nextstate name are the same as the SION of this signal. If it is the case, it changes the nextstate name as "Sink".

Submodule add_sink_state

This submodule is used to add a "Sink" state to the generated spec file.

– add_sink_state

It is the entry of this function. It simply applies the function create_sink_in_bloc to the parameter spec.

– create_sink_in_bloc

This function applies the function create_sink_in_process to each process of the process list in the block.

– create_sink_in_process

This function appends a new state "Sink" to the state list of the process.

– create_clause_list_for_sink

This function is used to create a clause list for the new state "Sink".

– create_clause_for_sink

This function creates Input clause for the state “Sink”.

– create_signal_in_for_sink

It creates input signal for the created Input Clause.

– create_stmt_for_sink

It creates an output statement for the Clause.

– create_signal_out_for_sink

It creates an output signal for the Output statement.

5.7 Conclusion

In this chapter, we have described CAML-SDL API and explained the implementation of the developed algorithms using CAML. For simplifying an original SDL specification and reducing the state (configuration) space, a set of experimental tools is developed for SDL machines. These tools allows one to perform a conservative abstraction in the sense that the state abstraction preserves the behavior of the original processes of the SDL specification (though new traces may appear). It may be useful for automatic test generation, verification and other related activities. In the next chapter, we apply these tools for a real SDL specification.

Chapter 6

Case Study

In this chapter we demonstrate how the developed tool set performs when applied to a relatively complex example SDL specification. We also use this example in order to evaluate (in an experimental way) how state abstraction techniques developed and implemented in this thesis affect the complexity of SDL specifications. To this end we execute our tools on the given SDL specifications and then we use the ObjectGeode simulator to calculate the total number of configurations (called states in the simulator) in both the original and transformed specifications to reveal any change in the number of configurations. Moreover, as the OG simulator reports other important properties of the simulated SDL specification, such as the configuration coverage rate, the number of transitions, and the transition coverage rate. These parameters characterize the completeness of the exhaustive simulation achieved by the OG tool when solving verification problems. The reason is that the exhaustive (complete) simulation usually requires testing the model under all possible inputs [Balc1995]. Combinations of feasible values of input signals and parameters can generate millions of logical paths in the model execution. Due to time, memory, and budgetary constraints, it is impossible to test the correctness of that many logical paths. So in practice, the exploration depth of simulation is bounded at least in order to avoid the storage memory overflow problem. As a result, the OG simulator cannot achieve a hundred percent coverage. Thus, the configuration coverage rate and the transition coverage rate achieved by the OG simulator implicitly characterize the complexity of the simulated SDL specification. One may safely conclude that if, exploring the abstracted system, the simulator reports a higher coverage compared to the original system then the process of abstraction has facilitated the exhaustive simulation with a chosen exploration depth.

6.1 The Original SDL Specification

The SDL specification, called mps.pr, the test example used in this work, describes a real protocol provided by France Telecom R&D. The file mps.pr contains about 2600 lines, so the SDL specification is not trivial. This specification has seventeen states, four input signals, four output signals, thirty-four variables (parameters) and three timers. Three input signals and one output signal are parameterized.

Table 6.1 Simulation Results for mps.pr

Exploration depth	**5**	**10**
Number of configurations	962	19517
Configurations coverage rate	62.50%	87.50%
Number of transitions	2698	50593
Transition coverage rate	38.71%	66.13%

Table 6.1 shows the simulation data for the original SDL specification mps.pr. It indicates, for example, that the OG simulator achieves just 87.50% coverage of the original SDL specification with the exploration depth limited to ten and 62.50% for the depth five.

6.2 Abstracting States

Next, we perform state abstraction of the specification. The original SDL mps.pr has contains the following seventeen states:

REPOS,
MENU_GRPVIDE,
MENU_GRPPLIN,
MENU_GRPNVNP,
MENU_DEBGROU,
MENU_NUMUNIQ,
MENU_ENRGNUM1,
MENU_ENRGNUM2,
MENU_EFFCONF,
MENU_GROUAUT,
MENU_FINGROU,
MENU_CONSUIT,
MENU_PRESSER,
MENU_NUMEFFA2,
MENU_NUMEFFA1,
ACCUEIL,
MENU_NUMENON

We assume that the user, by one reason or another, wants to merge them into twelve states as follows.

REPOS_ ACCUEIL,
MENU_GRPVIDE_MENU_GRPPLIN_MENU_GRPNVNP,
MENU_DEBGROU,
MENU_NUMUNIQ,

MENU_ENRGNUM1_MENU_ENRGNUM2,

MENU_EFFCONF,

MENU_GROUAUT,

MENU_FINGROU,

MENU_CONSUIT,

MENU_PRESSER,

MENU_NUMEFFA2_MENU_NUMEFFA1,

MENU_NUMENON

Here the notation REPOS_ ACCUEIL means that two states, REPOS and ACCUEIL are merged into a single state.

MENU_GRPVIDE_MENU_GRPPLIN_MENU_GRPNVNP comprises three states of the original specification; MENU_ENRGNUM1_MENU_ENRGNUM2 and MENU_NUMEFFA2_MENU_NUMEFFA1 have two states each.

We feed this data along with the SDL file mps.pr into our tool to merge states and obtain a transformed SDL specification. The SDL file is called mps_merge_label_decision.pr. Note that the observabilization tool is not applied at this stage. It will be used later.

Table 6.2 Simulation Results for mps_merge_label_decision.pr

Exploration depth	**5**	**10**
Number of configurations	2863	124570
Configurations coverage rate	78.95%	100.00%
Number of transitions	7526	506781
Transition coverage rate	62.50%	84.38%

Table 6.2 shows simulation data for the obtained SDL specification. Comparing Table 6.1 and Table 6.2, we can see that, for the same exploration depth limitation,

the OG simulator performs much better on the abstracted SDL specification than on the original file. Merging states allows to better exploring the SDL system.

6.3 Weak Observabilization of the Abstracted Specification

Our final step is to perform the observabilization step that also includes abstraction of a number of variables that need be removed and parameters. So we use all the tools developed in this work as well as the one of [Wang2001]. In particular, we first merge the states from 17 to 12 using our state abstraction tool. Then we unfold the labels and decisions in the original SDL specification by the developed tools label_process and decision_process. After that, we use our tool weak_observ_process to search and report parameters that need to be deleted. The tool reports that there are 23 parameters should be removed from the original specification. Finally all of the reported parameters are removed by means of the variable abstraction tool developed by [Wang2001]. Finally, we obtain a transformed SDL specification represented in the SDL file mps_merge_label_decision_abs_abs.pr. Table 6.3 presents the simulation data of the obtained SDL specification.

Table 6.3 Simulation Results for mps_merge_label_decision_abs_abs.pr

Exploration depth limited	**5**	**10**	**20**	**30**
Number of configurations	1657	5196	10189	42499
Configuration coverage rate	63.16%	94.74%	100.00%	100.00%
Number of transitions	3688	15226	29667	143099
Transition coverage rate	29.84%	60.88%	93.61%	100.00%

In this case, when we limit the exploration depth to thirty, we achieve a hundred percent state coverage and transition coverage and the exhaustive exploration of this system can be completed successfully. Notice that for both SDL specifications (see Table 6.1 and 6.2) we could not get a hundred percent state coverage and transition coverage for this exploration depth, and if we assume the same exploration depth, the simulation could not be completed because of memory overflow.

Several conclusions about the performance of our tool set and abstraction techniques could be drawn from this experiment.

Our experimental tools perform well on a pretty complex SDL specification (2600 lines). We do not use a special tool to monitor the exact run time of the tools; however, on this example, they require just under a minute to output the results. We can conclude that the experimental tools are sufficiently efficient. We can use the experimental tools developed in this thesis to simplify SDL specifications. SDL abstraction may be useful for automatic test generation, verification and other related activities.

The experimental results of Table 6.3 indicate that abstractions lead to a better specification coverage in simulation. Simulation is intended to find design errors, so the higher the coverage the more chances to reveal any design error in the system. There is, however, a price to pay here, as usual is. The abstracted system has a richer behavior than the original system, so the problem found in it does not necessarily appear in the original SDL system. It may be introduced into the system when the system is transformed (abstraction and observabilization).

At the same time, the experimental data demonstrate that the number of reachable configurations may sometimes increase when state abstraction is performed. This means that the effect of state abstractions in SDL should be evaluated on a case by case basis.

Chapter 7

Conclusions and Future Works

This research is devoted to the problem of making abstractions in FSM, EFSM and SDL. Our main conclusions from this work can be summarized as follows.

Abstraction techniques are usually applied to fight the so-called state explosion problem. In order to alleviate this problem some states of state machines can be merged. The resulting machine with fewer states can be considered as an abstraction of the given machine. In this work, we investigated the approach and the techniques for state merging in FSM, EFSM and eventually in SDL specifications. The main problem considered here is how to preserve the behavior and observability of the original machine. We proposed some solutions to this problem. To validate these solutions and proposed techniques we implemented a set of experimental tools for state abstraction and state observabilization. We demonstrated that the developed tools perform reasonably well on a real SDL specification and often allow one to simplify the original specification. The developed tool set can perform a required abstraction of the original SDL specification. It may be useful for automatic test generation, verification and other validation related activities.

Our results show that the proposed techniques in SDL specifications can often reduce the state (configuration) space. At the same time, they also demonstrate that the number of reachable configurations may sometimes increase when state abstraction is performed. The effect of state abstractions in SDL should be evaluated on a case by case basis. At the same time, the experimental results show that abstractions lead to a better specification coverage in simulation. This helps detecting design errors.

We used the implementation language Caml to implement the tool set. It is very powerful in several aspects and perfectly suited our goals. Our experience confirms that it serves well for rapid prototyping of advanced programs and experimental tool design.

Future works could focus on the following aspects. In this thesis, we implemented state merging and observabilization techniques so that an SDL specification can be observabilized provided that it meets the restrictions of the SDL-machine defined in Section 4.4. Functioning of SDL machines we defined is not formalized. To make the notion of an SDL machine more formal, more work is needed. On the other side, our tools do not treat SDL specifications that contain timers, saves, implicit saves (guarded inputs), priority inputs, spontaneous transitions, joins, and a few shortcuts such as state list, star symbols, input lists. Although we have discussed in this thesis rather general principles of state abstraction and observabilization, more implementation work is needed to further relax these constraints. It should not that difficult to remove some restrictions, which we used to simplify the problem, such as shortcuts; but more research is needed to remove those caused by the limitations of the Caml-SDL API.

References

[CCITT92] CCITT, COM X-R, 17-E, Geneva, March 1992, "Recommendation Z.100 – CCITT Specification and Description Language (SDL) and Annex A to the Recommendation.

[GEODE-1] VERILOG, "ObjectGeode – Tutorial, Version 4.0", VERILOG SA, France, April, 1999, http://www.verilog.fr

[GEODE-2] VERILOG, "ObjectGeode SDL Application Programming Interface-Reference Manual, Version 4.0", VERILOG SA, France, April, 1999, http://www.verilog.fr

[GEODE-3] VERILOG, "ObjectGeode SDL Simulator-Reference Manual", VERILOG SA, France, April, 1999, http://www.verilog.fr

[IS9074] "Estelle: A Formal Description Technique Based on an Extended State Transition Model", Int. Organization for Standardization, IS 9074, 1988.

[IS8807] "Information processing systems-Open System Interconnection-LOTOS – A Formal Description Technique Based on Temporal Ordering of Observed Behavior", Int. Organization for Standardization, IS 8807, 1988.

[Balc1995] O. Balci, "Principles and Techniques of Simulation Validation, Verification, and Testing", Proceedings of the 1995 Winter Simulation Conference, pp. 147-154, 1995.

[BH1989] F. Belina and D. Hogrefe, "The CCITT specification and description language SDL", Networks and ISDN Systems, 16, pp. 311-341, North-Holland, 1988/89.

[BP1994] G. v. Bochmann and A. Petrenko, "Protocol testing: Review of methods and relevance for software testing", Dep. Publ. #923, Universite de Montreal, 1994.

[BP+1997] G. v. Bochmann, A. Petrenko, O. Bellal, S. Maguiraga, "Automating the Process of Test Derivation from SDL Specification", Proc. 8th SDL Forum, France, Sept. 22-26, 1997, pp. 261-277.

[CW1996] E. M. Clarke, J. M. Wing, "Formal Methods: State of the Art and Future Directions", Computing Surveys 28(4), pp. 626-643, 1996.

[Gill1962] A. Gill, "Introduction to the Theory of Finite-State Machines", McGrawHill, New York, 1962.

[GS+1996] J. Grabowski, R. Scheurer, D. Toggweiler, and D. Hogrefe, " Dealing with the complexity of state space exploration algorithms for SDL systems", Arbeitsberichte des Instituts für mathematische Maschinen-und Datenverarbeitung (Mathematik), Proceedings of the 6th GI/ITG Technical Meeting on Formal Description Techniques for Distributed Systems, June 20 - 21, 1996, pp. 1-10, Vol. 20, No. 9, University of Erlangen, Germany, May 1996.

[HM1996] H. Higuchi and Y. Matsunaga, "A Fast State Reduction Algorithm for Incompletely Specified Finite State Machines", Proceedings of the 33rd annual conference on Design Automation, pp. 463 – 466, New York, June 1996.

[HK+2000] D. Hogrefe, B. Koch, and H. Neukirchen, "Validation and Testing", invited tutorial Teletronik, special issue on SDL, Telenor, 2000.

[Holz1991] G. Holzmann, "Design and Validation of Computer Protocols", Prentice-Hall, Englewood Cliffs, 1991.

[HM1999] N. Husberg, T. Manner, "Emma: Developing an Industrial Reachability Analyser for SDL", World Congress on Formal Methods (1) 1999, pp. 642-661.

[HU1979] J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley, 1979.

[KJ+1999] A. Kerbrat, T. Jeron, and R. Groz, "Automated Test Generation from SDL Specifications", SDL '99, the next millennium: proceedings of the ninth SDL Forum, Montréal, Québec, Canada, 21-25, pp. 135-151,1999.

[KV1992] N. Kumar and R. Vemuri. "Finite state machine verification on MIMD machines", In European Design Automation Conference, pp. 514-520, 1992.

[Lero2000] X. Leroy, "The Objective Caml System, release 3.00-Documentation and user's manual", Institute National de Researche en Informatique et en Automatique (INRIA), France, April 2000. http://www.inria.fr/

[LY1996] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines", Proceedings of the IEEE, pp. 1090-1123, August 1996.

[LC+1988] F.J. Lin, P. M. Chu, and M. T. Liu, "Protocol Verification Using Reachability Analysis: The State Space Explosion Problem and Relief Strategies", Proceedings of the ACM workshop on Frontiers in Computer Communication Technology, pp. 126-135, 1988.

[LG+1995] C. Loiseaux, S. Graf, J. Sifakis, A.Bouajjani, and S. Bensalem, "Property Preserving Abstractions for the Verification of Concurrent Systems", Formal Methods in System Design, 6, pp. 11-44, 1995.

[LP+1993] G. Luo, A. Petrenko and G. V. Bochmann, "Selecting Test Sequences for Partially-Specified Nondeterministic Finite State Machines", Dept. IRO, University of Montreal, Feb. 1993.

[LD+1994] G. Luo, A. Das and G. v. Bochmann, "Software testing based on SDL specifications with SAVE", IEEE Transactions on Software Eng., Vol. 20, 1, Jan. 1994, pp. 72-78.

[Marc1994] Marciniak, J.J. (Editor-in-Chief). Encyclopedia of Software Engineering. John Wiley and Sons, Inc., 1994.

[Maun1995] M. Mauny, "Functional Programming using Caml Light", Institute National de Researche en Informatique et en Automatique (INRIA), France, Jan., 1995. http://www.inria.fr/

[Monk1999] O. Monkewich, "SDL-based Specification and Testing Strategy for Communication Network", SDL '99, the next millennium: proceedings of the ninth SDL Forum, Montréal, Québec, Canada, 21-25, pp. 123-134, 1999.

[Oiko1996] K.N. Oikonomou, "Abstractions of Finite-state Machines and Optimality with Respect to Immediately-Detectable Next-State Faults", IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, pp. 151-160, Vol. 26, No.1, January 1996.

[PB1993] A. Petrenko, G. v. Bochmann, "Conformance Relations and Test Derivation", Proceedings of the IFIP 6th International Workshop on Protocol Test Systems, 1993, pp. 91-106, Pau, France.

[PY+1996] A. Petrenko, N. Yevtushenko, and G. v. Bochmann, "Testing Deterministic Implementations from Their Nondeterministic Specifications", Proceedings of the IFIP 9th International Workshop on Testing Communicating Systems, 1996, pp. 125-140.

[PB+1999] A. Petrenko, S. Boroday, and R. Groz, "Confirming Configurations in EFSM", IFIP TC6 WG6.2 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification, Oct. 1999, Beijing, China.

[SL1989] D. P. Sidhu, T. K. Leung, "Formal Methods for Protocol Testing: A Detailed Study", IEEE Trans. Software Eng. Vol. 15, No. 4, pp. 413-426, Apr. 1989.

[Star1972] P. H. Starke, "Abstract Automata", North-Holland/American Elsevier, 1972.

[TR1999] A. Touag, A. Rouger, "Methods and Methodology for an Incremental Test Generation from SDL Specifications", SDL '99, the next millennium:

proceedings of the ninth SDL Forum, Montréal, Québec, Canada, 21-25, pp. 153-168, 1999.

[TP+1995] Q. M. Tan, A. Petrenko and G. V. Bochmann, "Modeling Basic LOTOS by FSMs for Conformance Testing", Proceedings of the 15th International Symposium on Protocol Specification, Testing and Verification (PSTV 15), Poland, pp. 137-152, June 1995.

[Turn1993] K. J. Turner, "Using formal description techniques: an introduction to Estelle, Lotos, and SDL", Wiley, 1993.

[Vien1993] R. L. Vienneau, "Software Engineering", Computer Society Press, 1996.

[WL1993] C. J. Wang, M. T. Liu, "Generating Test Cases for EFSM with Given Fault Models", Proceeding of IEEE InFocom'93, Vol. 2, pp. 774-781, 1993.

[Wang2001] X. Wang, master thesis in progress, McGill University and CRIM, 2001.

Appendix A

Program for State Abstraction

Module state_merge_process

```
(******************************************************************************)
(*    Name:       state_merge_process.ml (main)                                *)
(*    Purpose:    Merging states of several blocks                             *)
(*    Author:     Hu Yun        Nov 30, 2000                                   *)
(*                                                                             *)
(*    Copyright (c) 2000 CRIM CANADA                                           *)
(******************************************************************************)
let main() =
  let arg = Sys.argv in
  let num = Array.length arg in
  if (num = 2) then
    let fname =
      if (Filename.check_suffix arg.(1) ".pr")
      then (Filename.chop_suffix arg.(1) ".pr")
      else arg.(1) in
    let e = Sdl.load 2 arg in
    match e with
      None ->
       begin
         print_endline("Echec du chargement de la specification "^arg.(1));
         exit 0
       end
    | Some _ ->
       begin
         let file_log = (fname^".log") in
         begin
           if (Sys.file_exists file_log) then Sys.remove file_log;
           Utils.set_log file_log;
            print_string "***\n";
           let spec0 = Extraction.extraction e file_log in
           print_string("\n\nThe number of states of the first process is ");
            print_int((State_num.state_num spec0)-1);
            print_string(State_name_list.state_name spec0);
            print_endline("\n\n");
            let file_log = (fname^"_merge"^".pr") in
             if (Sys.file_exists file_log) then Sys.remove file_log;
            Utils.set_log file_log;
            print_string "Enter the block number of this partition:";
            let state_list = Create_name_list.create_name_list spec0 in
            let block_number = read_int()
            in
            let merge = Array.create block_number "" in
            let block_list = Array.create block_number [] in
            for j = 0 to block_number-1 do
              print_string "Enter the state number of the ";
              print_int (j+1);
              print_string " block: ";
              let state_number = read_int() in
              let mergelist = Array.create state_number "" in
              let mergename = ref " " in
              for i = 0 to state_number-1 do
                print_string "Enter the name of the state:";
                let a = read_line() in
                if (List.mem a state_list) then
                    begin
                    if (i=0) then
                       mergename := a^(!mergename)
                    else
                       mergename := a^"_"^(!mergename);
                    mergelist.(i) <- a;
                    end
                else
                   print_endline("\n\nState Name Error: No state "^a^" in this SDL
spec file '"^fname^".pr' !!!\n\n");
              done;
              merge.(j) <- !mergename;
              block_list.(j) <- Array.to_list mergelist
            done;
            Utils.log (Spec_sdl.pretty (Process_merge.process_merge spec0 block_list
merge block_number));
            print_endline("A new SDL_spec file '"^fname^"_merge.pr' has been
generated!!!\n\n\n");
           exit 0
         end
      end
  else
    begin
      print_endline("State_Merge_process Error: Lack of Arguments!!!");
      exit 0
    end ;;

Printexc.catch main()

(*****************************************************************************)
(*    Module Name:           list_states.ml                                  *)
(*    Purpose:        construct a string with all state names                *)
(*****************************************************************************)
```

```
open Spec_sdl;;
let statelist list_states =

 let pretty_state state =
  match state with
   Start sl -> "\n\n "
  | Normal (s, cl) -> s^";" in
   String.concat "\n" (List.map pretty_state list_states)
```

```
(*****************************************************************************)
(*    Name:           state_name.ml                                          *)
(*    Purpose:        this function takes a spec and return state names      *)
(*****************************************************************************)
```

```
let state_name my_spec =
 let bl = my_spec.block in
  let ps = List.hd(bl.processes) in
    statelist ps.states;
```

```
(*****************************************************************************)
(*    Module Name:    state_name.ml                                          *)
(*    Purpose:        this function takes a spec and                         *)
(*                    return state number of a process                       *)
(*****************************************************************************)
```

```
open Spec_sdl;;
let state_num my_spec =
  let bl = my_spec.block in
    let ps = List.hd(bl.processes) in
             List.length ps.states;
```

Submodule process_merge

```
(*******************************************************************************)
(*    Name:       process_merge.ml                                            *)
(*    Purpose:    Merging states of several blocks                            *)
(*    Author:     Hu Yun                                                      *)
(*******************************************************************************)
open Spec_sdl;;
exception Error of string

(*******************************************************************************)
(*    Name:       rename_in_stmt                                              *)
(*    Purpose:    processing states in statement "NextState"                  *)
(*******************************************************************************)
let rec rename_in_stmt block_list merge_name block_number =
  function stmt ->
  match stmt with
  Terminator t -> (match t with
                   NextState n ->
                    (match n with
                       NoMinus l ->
                         let flag = ref 0 in
                         let num = ref 0 in
                           for i = 0 to block_number-1 do
                             if (List.mem l block_list.(i))
                             then
                               begin
                                 flag := 1;
                                 num := i
                               end
                           done;
                           if (!flag=1) then
                             Terminator (NextState (NoMinus (merge_name.(!num))))
                           else
                             stmt
                     | _ -> stmt)
                 | _ -> stmt)
| Decision d -> let rename_branche block_list merge_name block_number =
                 function (expr, ls_stmt) ->
                   (expr, List.map
                   (rename_in_stmt block_list merge_name block_number) ls_stmt)
                    in
                    Decision ({d with
                         branches =
                         List.map
                        (rename_branche block_list merge_name block_number)
                         d.branches;
                         branche_else =
                         List.map
                        (rename_in_stmt block_list merge_name block_number)
                         d.branche_else})
| _ -> stmt

(*******************************************************************************)
(*    Name:       rename_in_clause                                            *)
(*    Purpose:    parsing states in Clause                                    *)
(*******************************************************************************)
let rename_in_clause block_list merge_name block_number =
  function clause -> match clause with
                  Input ic -> Input ({ic with stmts =
                                     List.map (rename_in_stmt block_list
                                     merge_name block_number) ic.stmts})
                 | Input_star ic -> Input_star ({ic with stmts_star =
                                                 List.map
```

```
                                              (rename_in_stmt block_list
                                               merge_name block_number)
                                               ic.stmts_star})
                    | Cont_sig ic ->Cont_sig ({ic with stmts_cont_sig =
                                           List.map
                                          (rename_in_stmt block_list
                                           merge_name block_number)
                                           ic.stmts_cont_sig})
                    | _ -> clause

(*******************************************************************************)
(*    Name:       rename_in_state                                             *)
(*    Purpose:    processsing states in process or procudure                  *)
(*******************************************************************************)
let rename_in_state block_list merge_name block_number =
  function state -> match state with
                    Start stmt_list -> Start (List.map
                                          (rename_in_stmt block_list
                                           merge_name block_number)
                                           stmt_list)
                  | Normal (state_name, clause_list) ->
                                      let flag = ref 0 in
                                      let num = ref 0 in
                                      for i = 0 to block_number-1 do
                                        if (List.mem state_name block_list.(i))
                                        then
                                          begin
                                            flag := 1;
                                            num := i
                                          end
                                      done;
                                      if (!flag=1) then
                                        Normal (merge_name.(!num),
                                                List.map
                                               (rename_in_clause block_list
                                                merge_name block_number)
clause_list)
                                      else
                                       Normal (state_name,
                                                List.map
                                               (rename_in_clause block_list
                                                merge_name block_number)
clause_list)

(*******************************************************************************)
(*    Name:       process_clause_in_procedure                                 *)
(*    Purpose:    finding state list in a procedure                           *)
(*******************************************************************************)
let process_clause_in_procedure block_list merge_name block_number =
  function proc ->
 {proc with
  corps = List.map
         (rename_in_state block_list merge_name block_number)
          proc.corps}

(*******************************************************************************)
(*    Name:       process_in_clause_process                                   *)
(*    Purpose:    processing state list and procedure list in a process       *)
(*******************************************************************************)
let process_clause_in_process block_list merge_name block_number =
  function single_process ->
 {single_process with
  states = List.map
          (rename_in_state block_list merge_name block_number)
           single_process.states;
  procedures = List.map
              (process_clause_in_procedure block_list merge_name block_number)
               single_process.procedures}

(*******************************************************************************)
(*    Name:       process_clause_in_bloc                                      *)
(*    Purpose:    processing process list in a block                        *)
(***************************************************************************)
let process_clause_in_bloc single_bloc block_list merge_name block_number =
  {single_bloc with
   processes = List.map
              (process_clause_in_process block_list merge_name block_number)
               single_bloc.processes}

(***************************************************************************)
(*    Name:       process_merge                                             *)
(*    Purpose:    processing block in a spec                                *)
(***************************************************************************)
let process_merge spec block_list merge_name block_number =
  {spec with
   block = process_clause_in_bloc
           spec.block block_list merge_name block_number}
```

Appendix B

Programs for Flattening Decisions

Module decision_process

```
(*****************************************************************************)
(*    Name:       decision_process.ml (main)                                 *)
(*    Purpose:    flattening decision                                        *)
(*    Author:     Hu Yun        June, 2001                                   *)
(*    Copyright (c) 2001 CRIM CANADA                                         *)
(*****************************************************************************)
let main() =
  let arg = Sys.argv in
  let num = Array.length arg in
  if (num = 2) then
    let fname =
      if (Filename.check_suffix arg.(1) ".pr")
      then (Filename.chop_suffix arg.(1) ".pr")
      else arg.(1) in
    let e = Sdl.load 2 arg in
    match e with
      None ->
       begin
         print_endline("Echec du chargement de la specification "^arg.(1));
         exit 0
       end
    | Some _ ->
       begin
         let file_log = (fname^".log") in
         begin
           if (Sys.file_exists file_log) then Sys.remove file_log;
           Utils.set_log file_log;
           let spec0 = Extraction.extraction e file_log in
            let file_log = (fname^"_decision"^".pr") in
           if (Sys.file_exists file_log) then Sys.remove file_log;
            Utils.set_log file_log;
            Utils.log (Spec_sdl.pretty (Process_decisions.process_decisions spec0));
            print_endline("A new SDL_spec file '"^fname^
                          "_decision.pr' has been generated!!!\n\n\n");
           exit 0
          end
      end
 else
   begin
     print_endline("Decision process Error: Lack of Arguments!!!");
     exit 0
   end ;;
```

```
Printexc.catch main()
```

Submodule process_decisions

```
(*****************************************************************************)
(*    Name:       process_decisions.ml                                      *)
(*    Purpose:    flat decisions in a SDL_spec                              *)
(*    Author:     Hu Yun                                                    *)
(*****************************************************************************)
open Spec_sdl;;
exception Error of string

(*****************************************************************************)
(*    Name:       process_decision_merge                                    *)
(*    Purpose:    if there is still statement list after enddecision, append them *)
(*                to each branch of the decision                            *)
(*****************************************************************************)
let process_decision_merge tail =
  function branch -> ((fst branch), (List.append (snd branch) tail))

(*****************************************************************************)
(*    Name:       process_decision_in_stmt                                  *)
(*    Purpose:    process decisions in the Decision statement statement     *)
(*****************************************************************************)
let rec process_decision_in_stmt =
  function ic_stmts ->
    match ic_stmts with
      [] -> []
    | head::tail ->
        match head with
          Decision d ->
            if (tail==[]) then
              let rec process_decision_branche =
                function (expr, ls_stmt) ->
                  (expr, (process_decision_in_stmt ls_stmt)) in
                  [(Decision ({d with
                      branches = List.map process_decision_branche d.branches;
                      branche_else = process_decision_in_stmt d.branche_else}))]
            else
              if (d.branche_else = []) then
                let branch_list = List.map (process_decision_merge tail) d.branches in
                  let rec process_decision_branche =
                    function (expr, ls_stmt) ->
                      (expr, (process_decision_in_stmt ls_stmt)) in
                  [(Decision ({d with
                         branches = List.map process_decision_branche branch_list;
                         branche_else = process_decision_in_stmt d.branche_else}))]
              else
                let branch_list = List.map (process_decision_merge tail) d.branches in
                  let rec process_decision_branche =
                    function (expr, ls_stmt) ->
                      (expr, (process_decision_in_stmt ls_stmt)) in
                      [(Decision ({d with
                         branches = List.map process_decision_branche branch_list;
                         branche_else = process_decision_in_stmt
                                      (List.append tail d.branche_else)}))]
        | _ -> head::(process_decision_in_stmt tail)

(*****************************************************************************)
(*    Name:       process_decision_in_clause                                *)
(*    Purpose:    parsing states in Clause                                  *)
(*****************************************************************************)
let process_decision_in_clause =
    function clause ->
      match clause with
        Input ic ->
          Input ({ic with stmts = process_decision_in_stmt ic.stmts})
      | Input_star ic ->
          Input_star ({ic with stmts_star = process_decision_in_stmt ic.stmts_star})
      | Cont_sig ic ->
          Cont_sig ({ic with stmts_cont_sig =
                                      process_decision_in_stmt ic.stmts_cont_sig})
      | _ -> clause

(*****************************************************************************)
(*    Name:       process_decision_in_state                                  *)
(*    Purpose:    processsing states in process or procudure                 *)
(*****************************************************************************)
let process_decision_in_state =
  function state ->
    match state with
      Start stmt_ls ->
        Start (process_decision_in_stmt stmt_ls)
    | Normal (state_name, clause_list) ->
        Normal (state_name, (List.map process_decision_in_clause clause_list))

(*****************************************************************************)
(*    Name:       process_decision_in_procedure                              *)
(*    Purpose:    finding state list in a procedure                          *)
(*****************************************************************************)
let process_decision_in_procedure =
    function proc ->
      {proc with
       corps = List.map process_decision_in_state proc.corps}

(*****************************************************************************)
(*    Name:       process_decision_in_process                                *)
(*    Purpose:    processing state list and procedure list in a process      *)
(*****************************************************************************)
let process_decision_in_process =
  function single_process ->
   {single_process with
    states = List.map process_decision_in_state single_process.states;
    procedures = List.map process_decision_in_procedure single_process.procedures}

(*****************************************************************************)
(*    Name:       process_decision_in_bloc                                   *)
(*    Purpose:    processing process list in a block                         *)
(*****************************************************************************)
let process_decision_in_bloc single_bloc =
  {single_bloc with
   processes = List.map process_decision_in_process single_bloc.processes}

(*****************************************************************************)
(*    Name:       process_decisions                                          *)
(*    Purpose:    processing block in a spec                                 *)
(*****************************************************************************)
let process_decisions spec =
  {spec with
   block = process_decision_in_bloc spec.block}
```

Appendix C

Programs for Label Replacement

Module label_process

```
(*****************************************************************************)
(*    Name:       label_process.ml (main)                                    *)
(*    Purpose:    Flatting labels in the spec                                *)
(*    Author:     Hu Yun        June, 2001                                   *)
(*    Copyright (c) 2001 CRIM CANADA                                         *)
(*****************************************************************************)
let main() =
  let arg = Sys.argv in
  let num = Array.length arg in
  if (num = 2) then
    let fname =
      if (Filename.check_suffix arg.(1) ".pr")
      then (Filename.chop_suffix arg.(1) ".pr") else arg.(1) in
    let e = Sdl.load 2 arg in
    match e with
      None ->
       begin
         print_endline("Echec du chargement de la specification "^arg.(1)); exit 0
       end
    | Some _ ->
       begin
         let file_log = (fname^".log") in
         begin
           if (Sys.file_exists file_log) then Sys.remove file_log;
           Utils.set_log file_log;
           let spec0 = Extraction.extraction e file_log in
           let file_log = (fname^"_label"^".pr") in
           if (Sys.file_exists file_log) then Sys.remove file_log;
           Utils.set_log file_log;
           Process_label.process_label spec0;
           print_endline("A new SDL_spec file '"^fname^
                    "_label.pr' has been generated!!!\n\n\n");
         exit 0
       end
     end
   else
     begin
       print_endline("Label process Error: Lack of Arguments!!!");
       exit 0
     end ;;
```

```
Printexc.catch main()
```

Submodule process_label

```
(*****************************************************************************)
(*    Name:       process_label.ml                                          *)
(*    Purpose:    construct a list, each element of the list is composed    *)
(*                of a label name and a list followed this label            *)
(*    Author:     Hu Yun                                                    *)
(*****************************************************************************)
open Spec_sdl;;
exception Error of string

(*****************************************************************************)
(*    Name:       loop_detect                                               *)
(*    Purpose:    detecting loop in transition paths                        *)
(*****************************************************************************)
let rec loop_detect label flag =
  function ls_stmt ->
     match ls_stmt with
       [] -> flag
     | head::tail ->
         match head with
           Terminator t ->
             (match t with
                Join j ->
                  if (j=label)
                  then
                    begin
                      flag := 1;
                      print_string
                        ("\n\nA loop is detected on the label: "^label^"\n\n");
                    end
                  else ();
                  flag
              | _ -> (loop_detect label flag) tail)
         | Decision d ->
             let branchelse =
               if (d.branche_else = [])
               then (loop_detect label flag) tail
               else (loop_detect label flag) d.branche_else
             in
             let branch =
               (loop_detect label flag) (List.flatten (snd (List.split d.branches)))
             in
             if ((branchelse = ref 1) || (branch = ref 1)) then flag := 1 else ();
             flag
         | _ -> (loop_detect label flag) tail

(*****************************************************************************)
(*    Name:       process_label_in_stmt                                     *)
(*    Purpose:    processing label statement                                *)
(*****************************************************************************)
let rec process_label_in_stmt =
  function ic_stmts ->
    match ic_stmts with
      [] -> []
    | head::tail ->
        match head with
        Label l -> if (!((loop_detect l (ref 0)) tail)=1) then
                      (process_label_in_stmt tail)
                   else
                      (l, tail)
                       ::
                      (process_label_in_stmt tail)
      | Decision d -> let branchelse =
                      if (d.branche_else = [])
                      then
                        []
                      else
                        let else_str = process_label_in_stmt d.branche_else
                        in
                        else_str
                        in
                        let rec process_label_branche =
                          function (expr, ls_stmt) ->
                          let branch = process_label_in_stmt ls_stmt
                          in
                          if (branch = []) then [] else branch
                          in
                         List.append ((List.append
                                      (List.concat
                                      (List.map process_label_branche d.branches))
                                       branchelse)) (process_label_in_stmt tail)
      | _ -> process_label_in_stmt tail

(***************************************************************************)
(*    Name:       process_label_in_clause                                  *)
(*    Purpose:    process a signal clause and call the corresponding       *)
(*                statement list to function process_label_in_stmt         *)
(***************************************************************************)
let process_label_in_clause =
    function clause ->
      match clause with
        Input ic ->
          process_label_in_stmt ic.stmts
      |Input_star ic ->
         process_label_in_stmt ic.stmts_star
      |Cont_sig ic ->
         process_label_in_stmt ic.stmts_cont_sig
      |_ -> []

(***************************************************************************)
(*    Name:       process_label_in_state                                   *)
(*    Purpose:    processsing labels in process or procudure               *)
(***************************************************************************)
let process_label_in_state =
  function state ->
    match state with
      Start stmt_ls ->
        process_label_in_stmt stmt_ls
    | Normal (state_name, clause_list) ->
       List.flatten (List.map process_label_in_clause clause_list)

(***************************************************************************)
(*    Name:       process_label_in_process                                 *)
(*    Purpose:    processing process list in a block                       *)
(***************************************************************************)
let process_label_in_process spec =
  function single_process ->
      let label_list_in_process =
        List.flatten (List.map (process_label_in_state) single_process.states)
      in
      let label_pair = List.split label_list_in_process in
      let length = (List.length label_list_in_process ) in
      begin
      for i = 0 to length-1 do
        begin
        spec := Process_join_label.process_join_label !spec
               (fst (List.nth (label_list_in_process) i))
               (snd (List.nth (label_list_in_process ) i)) label_pair;
        end
      done;
      Utils.log (Spec_sdl.pretty !spec);
      print_endline("\n\n");
      end

(***************************************************************************)
(* Function name: process_label_in_bloc                                    *)
(* Parameter:  bloc, spec                                                  *)
(* Return: void                                                             *)
(* Purpose: apply process_label_in_process to each process of block         *)
(***************************************************************************)
let process_label_in_bloc spec =
  function single_bloc -> List.iter
                       (process_label_in_process spec)
                        single_bloc.processes

(***************************************************************************)
(* Function name: process_label                                             *)
(* Parameter:  spec                                                         *)
(* Return: void                                                             *)
(* Purpose: apply process_label_in_process to block                         *)
(***************************************************************************)
let process_label spec =
  (process_label_in_bloc (ref spec)) spec.block
```

Submodule process_join_label

```
(***************************************************************************)
(*    Name:       process_join_label.ml                                     *)
(*    Purpose:    remove labels in a SDL_spec                               *)
(*    Author:     Hu Yun                                                    *)
(***************************************************************************)
open Spec_sdl;;
exception Error of string

(***************************************************************************)
(*    Name:       process_tail_list                                         *)
(*    Purpose:    remove labels in the tail list                            *)
(***************************************************************************)
let rec process_tail_list label_list =
  function tail_list ->
    match tail_list with
      [] -> []
    | head::tail ->
        match head with
        Terminator t -> (match t with
                         Join j -> let l_list = fst label_list in
                                   let ll = ref 0 in
                                     for i = 0 to (List.length l_list)-1 do
                                       if (j = (List.nth l_list i))
                                       then ll := i
                                       else ();
                                     done;
                                    (process_tail_list label_list)
                                    (List.nth (snd label_list) !ll)
                       | _ -> head
                              ::
                            ((process_tail_list label_list) tail))
      | Decision d -> let rec process_join_label_branche label_list =
                        function (expr, ls_stmt) ->
                          (expr, ((process_tail_list label_list) ls_stmt))
                        in
                       (Decision ({d with
                                      branches =
                                      List.map
                                     (process_join_label_branche label_list)
                                      d.branches;
                                  branche_else =
                                 (process_tail_list label_list)
                                  d.branche_else}))
                        ::
                      ((process_tail_list label_list) tail)
      | Label l -> ((process_tail_list label_list) tail)
      | _ -> head
             ::
          ((process_tail_list label_list) tail)
```

```
(*****************************************************************************)
(*    Name:       process_join_label_in_stmt                                  *)
(*    Purpose:    remove labels in the Label statement and Terminator statement   *)
(*****************************************************************************)
let rec process_join_label_in_stmt label tail_list label_list =
  function ic_stmts ->
    match ic_stmts with
      [] -> []
    | head::tail ->
      match head with
      Terminator t ->
        (match t with
           Join j -> if (j = label)
                     then ((process_tail_list label_list) tail_list)
                     else head::((process_join_label_in_stmt
                                  label tail_list label_list) tail)
         | _ -> head::((process_join_label_in_stmt
                        label tail_list label_list) tail))
    | Decision d ->
        let rec process_join_label_branche label tail_list label_list =
          function (expr, ls_stmt) ->
            (expr,
            ((process_join_label_in_stmt label tail_list label_list) ls_stmt))
        in
        (Decision ({d with
                    branches = List.map (process_join_label_branche
                               label tail_list label_list)
                               d.branches;
                    branche_else = (process_join_label_in_stmt
                                    label tail_list label_list) d.branche_else}))
                                ::((process_join_label_in_stmt
                                    label tail_list label_list) tail)
    | Label l ->
        if (l = label)
        then ((process_join_label_in_stmt label tail_list label_list) tail)
        else head::((process_join_label_in_stmt label tail_list label_list) tail)
    | _ -> head:: ((process_join_label_in_stmt label tail_list label_list) tail)

(*****************************************************************************)
(*    Name:       process_join_label_in_clause                                *)
(*    Purpose:    parsing states in Clause                                    *)
(*****************************************************************************)

let process_join_label_in_clause label tail_list label_list =
    function clause ->
      match clause with
        Input ic ->
          Input ({ic with stmts =
                 (process_join_label_in_stmt label tail_list label_list)
                  ic.stmts})
      | Input_star ic ->
          Input_star ({ic with stmts_star =
                      (process_join_label_in_stmt label tail_list label_list)
                       ic.stmts_star})
      | Cont_sig ic ->
          Cont_sig ({ic with stmts_cont_sig =
                    (process_join_label_in_stmt label tail_list label_list)
                     ic.stmts_cont_sig})
      | _ -> clause

(*****************************************************************************)
(*    Name:       process_join_label_in_state                                 *)
(*    Purpose:    processsing states in process or procudure                  *)
(*****************************************************************************)
let process_join_label_in_state label tail_list label_list =
  function state ->
    match state with
      Start stmt_ls ->
        Start ((process_join_label_in_stmt label tail_list label_list)
                stmt_ls)
    | Normal (state_name, clause_list) ->
      Normal (state_name,
              (List.map
              (process_join_label_in_clause label tail_list label_list)
               clause_list))
(*****************************************************************************)
(*    Name:       process_join_label_process                                 *)
(*    Purpose:    processing state list and procedure list in a process      *)
(*****************************************************************************)
let process_join_label_in_process label tail_list label_list =
  function single_process ->
   {single_process with
    states = List.map
            (process_join_label_in_state label tail_list label_list)
             single_process.states}

(*****************************************************************************)
(*    Name:       process_join_label_in_bloc                                 *)
(*    Purpose:    processing process list in a block                         *)
(*****************************************************************************)
let process_join_label_in_bloc single_bloc label tail_list label_list =
  {single_bloc with
   processes = List.map
              (process_join_label_in_process label tail_list label_list)
               single_bloc.processes}

(*****************************************************************************)
(*    Name:       process_join_label                                         *)
(*    Purpose:    processing block in a spec                                 *)
(*****************************************************************************)
let process_join_label spec label tail_list label_list =
  {spec with
   block = process_join_label_in_bloc spec.block label tail_list label_list}
```

Appendix D

Programs for Weak Observabilization

Module weak_observ_process

```
(*****************************************************************************)
(*    Name:       weak_observ_process.ml (main   )                           *)
(*    Purpose:    observabilization of the spec                              *)
(*    Author:     Hu Yun        June, 2001                                   *)
(*    Copyright (c) 2001 CRIM, CANADA                                        *)
(*****************************************************************************)
open Spec_sdl;;

let main() =
  let arg = Sys.argv in
  let num = Array.length arg in
  if (num = 2) then
    let fname =
      if (Filename.check_suffix arg.(1) ".pr")
      then (Filename.chop_suffix arg.(1) ".pr")
      else arg.(1)
    in
    let e = Sdl.load 2 arg
    in
    match e with
      None -> begin
               print_endline("Echec du chargement de la specification "^arg.(1));
               exit 0
             end
    | Some _ -> begin
                 let file_log = (fname^".log") in
                   begin
                     if (Sys.file_exists file_log)
                      then Sys.remove file_log;
                     Utils.set_log file_log;
                     let spec0 = Extraction.extraction e file_log
                      in
                     print_string("\n\nThe number of states in the process is ");

                      print_int((State_num.state_num spec0)-1);
                      print_string(State_name_list.state_name spec0);
                      print_endline("\n\n");

                      let file_log = (fname^"_observ"^".pr")
                      in
                     if (Sys.file_exists file_log) then Sys.remove file_log;

                      Utils.set_log file_log;
```

```
                    Process_weak_observ.process_weak_observ spec0;
                    exit 0
                  end
              end
  else
    begin
      print_endline("Process Error: Lack of Arguments!!!");
      exit 0
    end ;;

Printexc.catch main()

(*****************************************************************************)
(*    Module Name:    list_states.ml                                     *)
(*    Purpose:        construct a string with all state names                *)
(*****************************************************************************)
open Spec_sdl;;
let statelist list_states =
  let pretty_state state =
    match state with
      Start sl ->
         "\n\n "
    | Normal (s, cl) ->
         s^";" in
      String.concat "\n" (List.map pretty_state list_states)

(*****************************************************************************)
(*    Name:           state_name.ml                                          *)
(*    Purpose:        this function takes a spec and return state names      *)
(*****************************************************************************)
let state_name my_spec =
  let bl = my_spec.block in
    let ps = List.hd(bl.processes) in
             statelist ps.states;

(*****************************************************************************)
(*    Module Name:    state_name.ml                                          *)
(*    Purpose:        this function takes a spec and                         *)
(*                    return state number of a process                       *)
(*****************************************************************************)
open Spec_sdl;;
let state_num my_spec =
  let bl = my_spec.block in
    let ps = List.hd(bl.processes) in
             List.length ps.states;
```

Submodule process_weak_observ

```
(*****************************************************************************)
(*    Module Name: process_weak_observ.ml                                    *)
(*    Purpose:     redirecting the nonobservable transitions to sink state   *)
(*    Author:      Hu Yun                                                    *)
(*****************************************************************************)
open Spec_sdl;;
open Var;;
exception Error of string

(*****************************************************************************)
(* Function name: process_stmt_in_path                                       *)
(* Parameter: stmt list                                                      *)
(* Return: stmt list list                                                    *)
(* Purpose:                                                                  *)
(*   process statement list in each path recursively to construct a list of stmt *)
(*   list of a transition path, the last stmt in the list is Nextstate.      *)
(*****************************************************************************)
let rec process_stmt_in_path =
  function stmts ->
    match stmts with
      [] -> []
    | head::tail ->
        match head with
        Decision d ->  let branchelse =
                         if (d.branche_else = []) then []
                         else
                           let else_str = process_stmt_in_path d.branche_else
                           in
                           else_str
                       in
                       let rec treat_branche =
                       function (expr, ls_stmt) ->
                       let branch = process_stmt_in_path ls_stmt
                       in
                       if (branch = []) then [] else branch
                       in
                       List.append (List.concat
                                   (List.map treat_branche d.branches))
                                    branchelse
      | Terminator t -> (match t with
                        NextState n -> (match n with
                                        NoMinus l -> [[head]]
                                      | _ -> [])
                      | _ ->[])
      | _ -> List.map (List.append [head]) (process_stmt_in_path tail)

(*****************************************************************************)
(* Function name: process_signal_in_path                                     *)
(* Parameter: signal, stmts                                                  *)
(* Return: stmt list list                                                    *)
(* Purpose:                                                                  *)
(*   process each input signal and pass the stmt list to the function        *)
(*   process_stmt_in_path.                                                   *)
(*****************************************************************************)
let process_signal_in_path stmts =
function signal -> process_stmt_in_path stmts

(*****************************************************************************)
(* Function name: process_clause_in_path                                     *)
(* Parameter: a clause                                                       *)
(* Return: stmt list list list                                               *)
(* Purpose:                                                                  *)
(*   process each clause, it apply the process_signal_in_path function the each  *)
(*   signal in the signal list and returns a stmt list list list in a path   *)
(*****************************************************************************)
let process_clause_in_path =
  function clause -> match clause with
                   Input ic -> List.map (process_signal_in_path ic.stmts) ic.signals
```

```
                  | _ -> []

(******************************************************************************)
(* Function name: treat_in_stmts                                              *)
(* Parameter: statement list                                                  *)
(* Purpose:                                                                   *)
(*   process statement list in each path recursively to construct a list of  *)
(*   output string list of a transition path, the last string in the list is *)
(*   the name of Nextstate.                                                   *)
(******************************************************************************)
let rec treat_in_stmts =
  function stmts ->
    match stmts with
      [] -> []
    | head::tail -> match head with
                    Output o -> List.map (List.append [o.sig_name])
                                         (treat_in_stmts tail)
                  | Terminator t -> (match t with
                                    NextState n -> (match n with
                                                    NoMinus l -> [[l]]
                                                  | _ -> [])
                                  | _ ->[])
                  | Decision d -> let branchelse =
                                    if (d.branche_else = []) then []
                                    else
                                      let else_str = treat_in_stmts d.branche_else
                                      in else_str
                                  in
                                  let rec process_branche =
                                    function (expr, ls_stmt) ->
                                    let branch = treat_in_stmts ls_stmt
                                    in
                                    if (branch = []) then [] else branch
                                    in
                                    List.append (List.concat
                                                (List.map process_branche d.branches))
                                                 branchelse
                  | _ -> treat_in_stmts tail

(******************************************************************************)
(* Function name:     treat_signal                                            *)
(* Parameter:         signal, stmt list                                       *)
(* Return:            stmt list list                                          *)
(* Purpose:           append a signal to the beginning of the statement list  *)
(******************************************************************************)
let treat_signal stmts =
function signal -> List.map
                  (List.append [signal.sig_name])
                  (treat_in_stmts stmts)

(******************************************************************************)
(* Function name:     treat_in_clause                                         *)
(* Parameter:         a clause                                                *)
(* Return:            string list list list                                   *)
(* Purpose:           process each clause, it returns string list list list   *)
(******************************************************************************)
let treat_in_clause =
  function clause -> match clause with
                    Input ic -> List.map (treat_signal ic.stmts) ic.signals
                  | _ -> []

(******************************************************************************)
(* Function name:     find_variable                                           *)
(* Parameter:         stmt                                                    *)
(* Return:            variable name                                           *)
(* Purpose:           to find the variable name in the left side of           *)
(*                    a Task statement                                        *)
(******************************************************************************)
let find_variable =
  function stmt -> match stmt with
                  Assign a -> (match a.left with
                                   Ident v -> v
                                 | _ -> "")
                    | _ -> ""


(******************************************************************************)
(* Function name:     treat_in_state                                          *)
(* Parameter:         state, spec                                             *)
(* Return:            void                                                    *)
(* Purpose:                                                                   *)
(*   1. for each state, construct state-input-output string list              *)
(*   2. for each state, construct stmt path list                              *)
(******************************************************************************)
let treat_in_state spec =
  function state -> match state with
                      Start stmt_list -> []
                    | Normal (state_name, clause_list) ->
             (*****************************************************************)
             (* for each state, construct a non-observable transition paths list *)
             (*****************************************************************)
                      let stmt_lll = ref [] in
                      let stmt_lll_array = ref [] in
                      let cl_list = List.map treat_in_clause clause_list in
                      let stmt_list_list = List.map process_clause_in_path clause_list
                      in
                      let stmt_list_flatten = List.flatten (List.flatten stmt_list_list)
                      in
                      let stmt_len = (List.length stmt_list_flatten) in
                      let cl_list_flatten_flatten = List.flatten (List.flatten cl_list)
                      in
                      let len = (List.length cl_list_flatten_flatten) in
                      let str_array = Array.create len "" in
                      let str_list_array = Array.create len [] in
                      let path_array = Array.create len "" in
                      let flag_array = Array.create len 0 in
                      print_string ("--------------------------------------------------
-----------------------\n");
                      print_string ("--------------------------------------------------
-----------------------\n");
                      print_string ("All transition paths from state: "^state_name^" ");
                      print_string "\n";
                      for i = 0 to len-1 do
                        let list_nth = state_name::(List.nth cl_list_flatten_flatten i)
                        in
                        let len_nth = (List.length list_nth) in
                        let array_nth = Array.create len_nth "" in
                        let array_nth = Array.of_list list_nth in
                          str_array.(i) <- (String.concat " " list_nth);
                        str_list_array.(i) <- list_nth;
                        print_string str_array.(i);
                        print_string "\n";
                        let path_array_nth = Array.create (len_nth-1) "" in
                        for j = 0 to len_nth-2 do
                          path_array_nth.(j)<-array_nth.(j);
                        done;
                        path_array.(i)<-
                                (String.concat " " (Array.to_list path_array_nth));
                      done;
                      print_string "\n";
                      print_string "\n";
                      for k = 0 to len-2 do
                        let flag = ref 0 in
                        let stmt_ll = ref [] in
                        let stmt_array = ref [] in
                        let list_ll = ref [k] in
                          flag_array.(k) <- 1;
                      for l = k+1 to len-1 do
                        if (path_array.(k) = path_array.(l)&&flag_array.(l) == 0)
                        then
                          begin
                            flag_array.(l) <- 1;
```

```
            stmt_ll := (List.nth stmt_list_flatten l)::(!stmt_ll);
            list_ll := l::(!list_ll);
          end
        else
          ()
      done;
(*********************************************************************)
(*  find the variable need to be delete                              *)
(*  construct a variable list                                        *)
(*  report each distinct variable                                    *)
(*********************************************************************)
    if (List.length (!list_ll)<>1) then
      begin
        stmt_ll := (List.nth stmt_list_flatten k)::(!stmt_ll);
        let all_variable_list = ref [] in
          for jj = 0 to (List.length (!stmt_ll))-1 do
            let stmt_l = (List.nth !stmt_ll jj) in
            all_variable_list := (List.map find_variable stmt_l)
                                   ::
                                 (!all_variable_list);
          done;

        let len_variable_list = List.length
                                (List.flatten (!all_variable_list))
        in
        let var_array = Array.create (len_variable_list) "" in
        let var_array = Array.of_list (List.flatten !all_variable_list)
        in
        for var1 = 0 to len_variable_list-2 do
          let sign = ref 0 in
          for var2 = var1+1 to len_variable_list-1 do
            if (var_array.(var1) =
              var_array.(var2)&&var_array.(var1)<>"")
            then
              sign :=1
            else
              ()
          done;
          if ((!sign)=0&&var_array.(var1)<>"") then
            begin
              print_string
                ("      Variable need to delete: "^var_array.(var1));
              print_string "\n";
            end
          else
          ();
        done;
        print_string "\n";
        print_string "      ";
        print_int (List.length (!stmt_ll));
        print_string " ";
        let len_ll = List.length (!list_ll)
        in
        print_string
        ("non-observable state-input-output-nextstate sequences
        found"^": \n");
        let same_end_flag = ref 0 in
        let identical_str = str_array.(List.nth (!list_ll) 0) in
        for ll = 1 to len_ll-1 do
          if (identical_str <> str_array.(List.nth (!list_ll) ll))
          then same_end_flag := 1
        done;
        for ll = 0 to len_ll-1 do
          begin
            print_string "      ";
            print_string str_array.(List.nth (!list_ll) ll);
            print_string "\n";
            if (!same_end_flag = 1)
            then
              stmt_lll_array :=
                (Array.of_list
                            (str_list_array.(List.nth (!list_ll) ll)))
                            ::
                            (!stmt_lll_array);
                         end
                       done;
                       print_string "\n";
                       print_string "\n";
                     end
                   else
                     ();
                 done;
               (!stmt_lll_array)

(****************************************************************************)
(* Function name:             process_clause_in_process                     *)
(* Parameter:                 spec                                          *)
(* Return:                    void                                          *)
(* Purpose:                   apply treat_in_state to each process          *)
(****************************************************************************)
let process_clause_in_process spec =
  function single_process ->
    begin
      let nonobe_list = List.flatten
                        (List.map (treat_in_state spec) single_process.states)
      in
      let nonobe_len = List.length nonobe_list in
      let result_list = ref [] in
      for j = 0 to nonobe_len-1 do
        if ((List.nth nonobe_list j).(0) == "") then
          ()
        else
          result_list := (List.nth nonobe_list j)::(!result_list);
      done;
      let length = (List.length !result_list) in
      for i = 0 to length-1 do
        begin
        spec := Process_redirect_to_sink.process_redirect_to_sink
                !spec
               (List.nth (!result_list) i);
        end
      done;
      spec := Add_sink_state.add_sink_state !spec;
      Utils.log (Spec_sdl.pretty !spec);
      print_endline("\n\n");
    end

(****************************************************************************)
(* Function name:             process_clause_in_bloc                        *)
(* Parameter:                 bloc, spec                                    *)
(* Return:                    void                                          *)
(* Purpose:                   apply process_clause_in_process to each block *)
(****************************************************************************)
let process_clause_in_bloc spec =
  function single_bloc -> List.iter (process_clause_in_process spec)
                                    single_bloc.processes

(****************************************************************************)
(* Function name:             process_weak_observ                           *)
(* Parameter:                 spec                                          *)
(* Return:                    void                                          *)
(* Purpose:                   apply process_clause_in_process to block      *)
(****************************************************************************)
let process_weak_observ spec = (process_clause_in_bloc (ref spec)) spec.block
```

Submodule process_redirect_to_sink

```
(******************************************************************************)
(*    Name:       process_redirect_to_sink.ml                                 *)
(*    Purpose:    redirecting nonobservable transition path to sink state and *)
(*                deleting all tasks in this transition path except the output *)
(*                statement                                                   *)
(*    Author:     Hu Yun                                                      *)
(******************************************************************************)
open Spec_sdl;;
exception Error of string

(******************************************************************************)
(*    Name:       create_signal_out                                           *)
(*    Purpose:    creating a output signal                                    *)
(******************************************************************************)
let create_signal_out =
  function msg_out ->
    {sig_name = msg_out; sig_params = []}

(******************************************************************************)
(*    Name:       create_output_stmt                                          *)
(*    Purpose:    creating a output statement                                 *)
(******************************************************************************)
let rec create_output_stmt =
  function msg_out -> [Output (create_signal_out msg_out)]

(******************************************************************************)
(*    Name:       redirect_in_clause                                          *)
(*    Purpose:    for each clause, if the input name is in the SION sequence, *)
(*                it creates a new stmt list for this input statement         *)
(*                which ignore all other statement in the transition path except *)
(*                the output statement.                                       *)
(*                at last, it appends a 'sink' Nextstate Terminator to        *)
(*                the stmt list                                               *)
(******************************************************************************)
let redirect_in_clause array_list input_name =
  function clause ->
  match clause with
  Input ic -> if (input_name == ((List.hd ic.signals).sig_name)) then
                Input ({ic with stmts =
                       (List.append
                       (List.flatten
                       (List.map  create_output_stmt (Array.to_list array_list)))
                       [Terminator (NextState (NoMinus "sink"))])
                       })
            else
              clause
| _ -> clause

(******************************************************************************)
(*    Name:       treat_signal                                                *)
(*    Purpose:                                                                *)
(******************************************************************************)
let treat_signal ic =
  function signal -> Input ({ic with signals = [signal]})

(******************************************************************************)
(*    Name:       process_clause                                              *)
(*    Purpose:    reconstruct clause list                                     *)
(******************************************************************************)
let process_clause =
  function clause ->
  match clause with
  Input ic -> if ((List.length ic.signals) > 1) then
                List.map (treat_signal ic) ic.signals
              else
               [clause]
| _ -> [clause]
```

```
(******************************************************************************)
(*    Name:       redirect_in_state                                          *)
(*    Purpose:    processsing states in process or procudure                 *)
(******************************************************************************)
let redirect_in_state array_list =
  function state ->
  match state with
  Start stmt_ls -> state
| Normal (state_name, clause_list) ->
    if (array_list.(0) == state_name) then
        let len = Array.length array_list in
        let array_new = Array.create (len-3) "" in
          begin
            for i = 0 to (len-4) do
              array_new.(i) <- array_list.(i+2);
            done;
          end;
       Normal (state_name, List.map (redirect_in_clause array_new array_list.(1))
                          (List.flatten (List.map process_clause clause_list)))
    else
        state

(******************************************************************************)
(*    Name:       process_clause_in_procedure                                *)
(*    Purpose:    finding state list in a procedure                          *)
(******************************************************************************)
let process_clause_in_procedure array_list =
  function proc -> {proc with
                    corps = List.map (redirect_in_state array_list) proc.corps}

(******************************************************************************)
(*    Name:       process_in_clause_process                                  *)
(*    Purpose:    processing state list and procedure list in a process      *)
(******************************************************************************)
let process_clause_in_process array_list =
  function single_process ->
   {single_process with
   states = List.map (redirect_in_state array_list) single_process.states;
   procedures = List.map (process_clause_in_procedure array_list)
                         single_process.procedures}

(******************************************************************************)
(*    Name:       process_clause_in_bloc                                     *)
(*    Purpose:    processing process list in a block                         *)
(******************************************************************************)
let process_in_bloc single_bloc array_list =
  {single_bloc with
   processes = List.map (process_clause_in_process array_list)
                        single_bloc.processes}

(******************************************************************************)
(*    Name:       process_redirect_to_sink                                   *)
(*    Purpose:    processing block in a spec                                 *)
(******************************************************************************)
let process_redirect_to_sink spec array_list =
  {spec with
   block = process_in_bloc spec.block array_list}
```

Submodule add_sink_state

```
(*****************************************************************************)
(*    Name:       add_sink_state.ml                                         *)
(*    Purpose:    add a sink state in the SDL spec                          *)
(*    Author:     Hu Yun                                                    *)
(*****************************************************************************)
open Spec_sdl;;
exception Error of string

(*****************************************************************************)
(*    Name:       create_signal_out_for_sink                                *)
(*    Purpose:    creating a output signal                                  *)
(*****************************************************************************)
let create_signal_out_for_sink =
  function msg_out ->
    {sig_name = msg_out; sig_params = []}

(*****************************************************************************)
(*    Name:       create_stmt_for_sink                                      *)
(*    Purpose:    creating a output statement                               *)
(*****************************************************************************)
let rec create_stmt_for_sink =
  function msg_out -> [Output (create_signal_out_for_sink msg_out)]

(*****************************************************************************)
(*    Name:       create_signal_in_for_sink                                 *)
(*    Purpose:    creating an input signal                                  *)
(*****************************************************************************)
let create_signal_in_for_sink =
  function msg_in ->
    [{sig_name = msg_in; sig_params = []}]

(*****************************************************************************)
(*    Name:       create_clause_for_sink                                    *)
(*    Purpose:    creating an Input clause                                  *)
(*****************************************************************************)
let create_clause_for_sink msg_in =
  function msg_out -> Input {signals = create_signal_in_for_sink msg_in;
                            garde_opt = None;
                            stmts = List.append
                                   (create_stmt_for_sink msg_out)
                                   [Terminator (NextState (NoMinus "sink"))];
                            parametres_non_signifiants = false}

(*****************************************************************************)
(*    Name:       create_clause_list_for_sink                               *)
(*    Purpose:    creating a clause list                                    *)
(*****************************************************************************)
let create_clause_list_for_sink msg_out =
  function msg_in -> List.map (create_clause_for_sink msg_in) msg_out

(*****************************************************************************)
(*    Name:       create_sink_in_process                                    *)
(*    Purpose:    adding a new state "sink" to state list of a process      *)
(*****************************************************************************)
let create_sink_in_process msg_out =
  function single_process ->
   {single_process with
    states = List.append
             single_process.states [(Normal ("sink",
                   List.flatten
                  (List.map
                  (create_clause_list_for_sink msg_out)
                   single_process.messages_in)))] }

(*****************************************************************************)
(*    Name:       create_sink_in_bloc                                       *)
(*    Purpose:    processing process list in a block                        *)
(*****************************************************************************)
```

```
let create_sink_in_bloc msg_out =
  function single_bloc ->
  {single_bloc with
   processes = List.map (create_sink_in_process msg_out)
                          single_bloc.processes}

(*******************************************************************************)
(*    Name:       add_sink_state                                               *)
(*    Purpose:    processing block in a spec                                   *)
(*******************************************************************************)
let add_sink_state spec =
  {spec with
   block = (create_sink_in_bloc spec.messages_out) spec.block}
```