Techniques for Abstracting SDL Specifications

Sergiy Boroday[1], Roland Groz[2], Alex Petrenko[1], Yves-Marie Quemener[2]

1 –*CRIM, Centre de Recherche Informatique de Montréal*
550 Sherbrooke West, Suite 100, Montreal, H3A 1B9, Canada
phone: +1 (514) 840-1234, fax: +1 (514) 840-1244,
Boroday@crim.ca, Petrenko@crim.ca

2 – *France Telecom R&D*
France Télécom - R&D
2 av. Pierre Marzin - F-22307 Lannion cedex, France
phone: +33 2.96.05.37.90, fax: +33 2.96.05.39.45,
Roland.Groz@francetelecom.com, yvesmarie.quemener@rd.francetelecom.com

Abstract: Abstracting the behaviour of a specification is a key technique for dealing with the complexity of such tasks as reachability analysis and test generation. We adapted classical data-flow analysis techniques to abstract variables in SDL processes and addressed the problem of finding conservative state abstractions. Prototype tools have been developed to implement those techniques and applied to simple applications from the field of telecommunications.

Key words: communicating extended state machines, SDL, fault models, data-flow, abstraction.

1. INTRODUCTION

Validation activities in software development based on SDL are usually faced with the high complexity. Multi-process asynchrony, rich data and complex control structures reflect the intrinsic complexity of such systems. Correspondingly, the state space of the global system is so huge that most software analysis tasks that need to resort to some sort of reachability analysis or state exploration are faced with the state explosion problem [GS+96].

The work reported here was initially motivated by the need to address this complexity issue in the case of a test generation method based on the underlying EFSM semantics of SDL specifications. Our test generation method takes into account both the control part and the data part of the SDL model and tries to cover both the structure of the specification and faults that can affect the internal data of a system. In our test generation approach [PBG99][BP+02], we try to generate test sequences as discriminating sequences between a reference SDL specification and faulty implementations, which can be modelled by classes of mutated SDL descriptions that constitute what is called a fault model [BD+91][PeY92][Pet00]. Computation of the test sequences relies on a combined reachability analysis of the SDL specification and of a representation of the fault model.

The complexity of the fault model is therefore a key issue. In order to tackle state explosion, we investigated several abstraction techniques.

Since our fault model includes both faults on the control structure and in the data part of SDL, we considered two kinds of abstraction techniques, which we call variable and state abstraction.

Abstractions to reduce complexity

An abstraction is a simplified description, or specification, of a system that emphasises some of the system's details or properties while suppressing others. A good abstraction is one that emphasises details that are significant for the purpose of the user (or sometimes just for the reader) and suppresses details that are, at least temporarily, immaterial or diversionary. In particular, abstraction is widely used in model checking and verification of both hardware and software systems. In [MAH98], abstraction is used to capture the control flow of the original circuit for redundancy identification and test generation at the state transition level. One of three abstraction techniques mentioned in [CGP99] is variable abstraction. The technique explained there represents a method of performing abstraction different from what we need to do here. Instead of removing non-relevant variables, it replaces a variable that ranges over a large (often infinite) set of values with a more abstract discrete-valued variable.

A theoretical framework for abstraction in computer languages has been provided by the seminal work of Cousot [Cou77]. Such an approach has been applied in [LG+95] to property preserving transformations for reactive systems. A key idea is the use of Galois connection and (α, γ)-simulation which is the same as the standard simulation often used to define implementation. Although we did not reuse directly that semantic framework, we have worked on what we call conservative abstractions, i.e. abstractions where the observable input-output behaviours of the abstract machine constitute a superset of the behaviours of the original specification; allowing suppression of some output parameters (since parameters are less important details of behaviour than output signals). Assuming that missing parameters have "don't care" values, the abstraction is conservative if and only if the original conforms to the abstraction.

Related practical results on abstraction are widely known in program slicing [Tip95]. However, here we consider deeper abstractions that could be achieved by allowing non-determinism, which is common for specifications, and is not supported in traditional programming languages.

Abstraction in SDL

Variable abstraction on SDL for model checking properties has already been addressed by [Boz99]. In [BFG99], a first technique based on data-flow analysis similar to the one we used is applied to reduce the state vector in the model checker. [Boz99] also uses domain analysis to reduce the state space.

Although variable abstraction had already been studied for SDL, to the best of our knowledge, state abstraction is a new contribution of this paper. In fact, our work is an attempt at extending FSM abstraction techniques to the much more complex structure of SDL.

According to [Oik96], 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. [Oik96] 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.

Whereas our work fits into that framework, we depart from [Oik96] in two respects. First, unlike [Oik96], we do not merge inputs and outputs. This stems from our initial need: from a testing point of view, a class of faulty behaviours can be collapsed to a simpler, more non-deterministic control structure, but the outputs and inputs remain distinct. The main difference however is that we deal with SDL processes, and not just with FSM.

In our work on SDL abstraction and test generation, we have used the Object Geode environment, mostly the Object Geode API interface to the parser, and the verifier for reachability analysis. Actually, our test generation approach was first motivated by the need to enhance the capabilities of the TestComposer generation tool and the approach that had been presented in [KJG99]. This may account for some of the peculiarities of our tools, as a few choices were made to take into account the features of that environment.

The rest of this paper is organised as follows. In Section 2, we present how we implemented data-flow techniques to perform abstraction on SDL specifications. Section 3 is devoted to state abstraction. As it deals with the state-oriented structure of SDL processes, we first define it in an EFSM theoretical framework. Then we discuss how all the features of the SDL language could interfere with abstraction, and propose solutions to address abstraction of specifications including most SDL constructs. Prototype tools that implement the abstraction are also briefly presented. In Section 4, we give a few results on the reduction of complexity that was reached on a typical case study, before concluding in Section 5.

2. VARIABLE ABSTRACTION

In this section, we discuss the problem of abstraction (removal) of a set of variables from an SDL specification. Currently, we restrict ourselves to abstracting variables within one process at a time. However, extending dependencies to take into account inter-process communications should not be a major problem, since our approach is based on static analysis.

Taking into account the fact that the variables to be removed can define other variables, the list of destroyed variables has to be extended by including each variable that depends on a variable already in the list. In the extreme case, all the variables can be deleted from the given process. Note that if an output parameter depends on a variable to be deleted, then it has also to be removed from the process, as its value cannot be uniquely determined. A decision that depends on a deleted variable become non-deterministic and should be replaced by an ANY decision. The remaining variables, parameters, and modified decisions define an SDL projection.

Please note that contrary to what has often been done for reachability analysis of SDL specifications (e.g. for symbolic model checking), the abstraction is done on variables that may change the behaviour of the specification. In fact, an abstraction corresponds to a class of faulty implementation in our fault model. It is a convenient mean to represent in a single SDL form a set of erroneous implementations (mutants) of our original specification.

Since we simulate concurrently the reference SDL specification and the abstracted one, our test generation method ends up with fully instantiated scenarios, because we get values for the abstracted variables from the simulation of the reference SDL specification [BP+02] [PBG99].

The initial variables to be abstracted from would typically be produced by our fault model, or more precisely the mutants we use for our test generation. Variable abstraction is used as a preliminary step to determine the "cone of dependence" of the variables that depend on those initial variables, and compute the resulting SDL model of the fault class.

2.1 Variable Dependence in SDL Language

Abstracting variables of an SDL process amounts to erasing (projecting out) some variables along with their declaration. However, only a consistent set of variables, on which no other variables depend, may be deleted correctly. Otherwise, the resulting SDL process could be syntactically or semantically incorrect.

Definition. Let *a* and *b* be two variables of an SDL process. We say that *a directly depends* on *b* if

1) *a* is in the left-hand part of an assignment expression and *b* is in the right-hand part or
2) *b* is used in an actual parameter of a procedure call and *a* is a corresponding formal parameter.

Variables *b* and *a* are said to be in the *direct dependence* relation.

The direct dependence relation can be constructed by inspecting the SDL specification (its abstract syntax tree). The transitive closure of the direct dependence relation is the *dependence* relation (recall that the transitive closure is the minimal transitive relation that contains the original relation).

Let v_1 be a variable to be deleted. Then the closure of the set $\{v_1\}$ under the dependence relation is a minimal set $[\{v_1\}]$ that contains all variables, which depend on v_1 (are defined by v_1). So if we intend to delete v_1 we should delete set $[\{v_1\}]$. As an example, assume that we need to delete the variable *D* in an SDL process that has the following assignments

A := *D* + *C*
B := *A* + 1
B := *C*
D := *C* - 1

As can be seen from the dependence relation graph shown in Figure 1, the closure of $\{D\}$ is the set $[\{D\}] = \{A, B, D\}$.

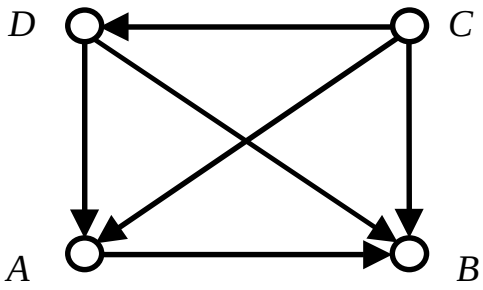

Figure 1: The dependence relation graph.

We use an algorithm for computing the dependence relation, based on a breadth-first search [CLR92]. Since the direct dependence graphs are usually sparse and relatively small it is a satisfactory solution, though, the Floyd-Warshall algorithm for the transitive closure [CLR92] might be more effective in a more general setting.

2.2 Projection in SDL

In SDL the projection of the closed under the dependence relation set of variables is performed by removing these variables, their assignments and declarations. The decisions, which use these variables, are replaced with ANY decisions. As an approximation, all guarding conditions (predicates) and continuous conditions with these variables are replaced with the True expression. The latter transformation could be non-conservative when the process may refuse to consume a signal from its input queue (it contains the SAVE construct or an enabling condition). A conservative approximation would require replacing such predicates with an ANY (Boolean). Unfortunately, such a conservative approximation results in a specification which could not be simulated by existing tools, and results in a larger state space. Since parameters are optional in SDL we remove only expressions, used as parameters in output actions.

We implement abstraction of an SDL specification using its abstract syntax tree. The extraction of the abstract syntax tree is performed with the Object Geode SDL-92 Application Programming Interface (SDL API).

The SDL API offers access to all the objects of the internal SDL data structure. Instead of creating new data structures for the variable abstraction, we just use the internal data structures that the API offers. The process involves the following steps: extraction of the abstract syntax tree, dependence analysis, abstraction on the tree (by marking elements to be deleted or modified), and pretty-printing of the marked tree into the SDL specification. While pretty-printing uses a complete abstract syntax tree, the abstraction deals only with the following objects of the tree:

- Entity (system, block, process, substructure, procedure, etc.).
- Var (variable, formal parameter, newtype, etc.).
- State (syntactic state).
- Clause.
- Stmt (task assignment, decision, output, export, if, while, etc.).
- Expr (all kinds of expressions).

Based on the relationship of these objects, a tool performs their marking [Wan01]. In particular, a variable to be removed along with all the variables that depend on it are marked as to be deleted. All the assignment, procedure and output statements, where any of those variables appears are also marked in this way. The tool also marks each decision statement, enabling condition and continuous signal that involves a marked variable as to be modified.

To process the marked tree, we use a program from the Object Geode toolbox that supports the pretty-printing of the abstract syntax tree into SDL-like form. We only modify this program in order to print out the abstract syntax tree into a valid SDL specification taking into account the markings. Any marked “to be removed” element is not printed out, while each marked decision is replaced by ANY decision and each marked enabling condition as well as continuous signal is replaced by True.

Some variables, for which we have not yet developed dependence analysis and abstraction methods, are exempted from abstraction (along with dependent variables). These are shared variables, variables used in actual IN-OUT parameters of procedures, variables used in timer setting. This comes from the fact that we built our abstraction computation on a relatively simple EFSM model for SDL processes, so we did not address the full semantics of the language. Thus, since we concentrate on single process abstraction and do not perform inter-process dependence analysis, we consider the shared variables in SDL as irremovable and keep them untouched to avoid changing the behaviour of the concerned SDL processes. IN-OUT parameters are subject to ‘side effects’, which are difficult for analysis, so they are considered irremovable. Variables, on which irremovable variables depend on, are considered to be irremovable themselves.

Variables and expressions could be used as actual parameters of the procedure calls. For the sake of simplicity, formal parameters of a procedure are treated as variables. Appearance of an expression in procedure call is considered as an assignment of a corresponding formal parameter.

3. STATE ABSTRACTION

In this section, we consider abstraction techniques dealing with the states of an SDL process. The abstraction consists in merging states, so that the number of states is reduced, and the structure of the control automaton can be simplified, because redundant transitions might be merged as well. Just as in the case of variable abstraction, we consider here that the level of abstraction to be performed is defined prior to applying abstraction techniques. The starting hypothesis is that we are given a partition of the set of states. The different states of a given class of this partition are considered to be semantically indistinguishable from that abstract point of view. For instance, in our

test generation approach [PBG99], a class could correspond to a given initialisation or transfer fault in a fault model: an erroneous implementation might reach any member of that class, instead of the specified reference state.

Merging states for abstraction is an extension of the reduction of a non-minimal machine through the removal of redundant states. The basic method for machine reduction by successive merging equivalent states was given in [Gil62]. This method emphasised on merging simply equivalent pair of states successively to get a reduced machine of the original one. State minimisation for the EFSM model let alone for SDL is a much harder problem, for which no efficient technique exists; in fact, there is no clear goal of what should be minimised, given that the variables can encode state information. 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. However, as state abstraction is more easily discussed in the framework of the automaton structure of SDL, we first give definitions for EFSM.

3.1 Extended Finite State Machine

The EFSM model we need must include multiple output signals, as this makes it easier to map SDL into this model. Here, we extend the model we had used in [PBG99].

Definition. 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 tuple (s, x, P, op, w, up, s'), where

- $s, s' \in S$ are the initial and final states of the transition, respectively;
- $x \in X$ is input, X is a finite set of inputs, and each $x \in X$ is associated with a set of input vectors D_{inp_x}, each component of an input vector corresponds to an input parameter associated with x, the sets of input parameters of different signals are mutually disjoint;
- $w \in Y^*$ is a sequence of output signals $y_1y_2...y_n$, $y_i \in Y$, where Y is a finite set of (elemental) outputs, disjoint from X, each $y \in Y$ is associated with a set of output vectors D_{out_y}, each component of an output vector corresponds to an output parameter associated with y, the sets of output parameters of different signals are mutually disjoint; 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: $D_{inp_x} \times D_V \rightarrow D_{out_{y_1}} \times D_{out_{y_2}} \times ... \times D_{out_{y_n}}$ is an output parameter function that defines values of parameters of each signal in the output word;
– up: $D_{inp_x} \times D_V \rightarrow D_V$ is a context update function.

We normally use (s—x,P/op,w,up→s') to denote a transition $t \in T$, we will also use the notation (s—x,P/y_1(op_1)y_2(op_2)...y_n(op_n),up→s'). An empty output word will be dropped from a transition to simplify its notation and such a transition is called a *silent* transition.

As an example, the EFSM in Figure 2 has four states, two integer context variables, two inputs a and b, three outputs x, y, z, the latter is parameterised with an integer parameter, and four transitions are guarded with predicates different from True.

A *parameterised* signal, input or output, is a pair of the signal itself and a vector of its parameters. It may also be defined as a pair of a signal and a mapping of its parameters into the parameter values. We will use notations for signals in the form of, for example, y(2, 3) meaning the signal y whose two parameters are instantiated to 2 and 3.

Definition. A context vector $\vec{v} \in D_V$ is called a *context* of M. A *configuration* of M is a pair of state s and context $\vec{v}$.

A context may also be defined as a mapping of variables into variable values.

In the case of empty sets of context variables and parameters, we do not make a difference between configuration and state, in other words, the EFSM model includes the FSM model as a special case.

The EFSM operates as defined in [PBG99]. The machine usually starts from a designated configuration, called the *initial* configuration. A pair of an EFSM and the initial configuration is called an *initialised* EFSM. The *behaviour* of an initialised EFSM is the set of all its input-output sequences.

Definition. An *input-output sequence* of an EFSM (or input-output word) is a sequence of parameterised inputs interleaved by parameterised outputs produced by the EFSM executing a finite sequence of transitions from a certain configuration in response to these inputs.

We target a class of specification EFSMs which are consistent, completely specified, deterministic, and observable [PBG99]. EFSM is called *observable* if in the underlying FSM, each transition can be uniquely identified by its input, output, and source state.

3.2 Abstracting States in the EFSM Model

Abstraction is defined as follows.

Definition. 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. An EFSM over the same alphabets is called a *factor* EFSM for A, denoted A_π if each state of A_π is a class of the partition π, the initial state of EFSM belongs to initial state of *factor* EFSM, and, for each transition $(s—x,P/op,w,up→s')$ of A, there exists a transition $(\pi(s)—x,P/op,w,up→\pi(s'))$ and vice versa.

The following characterises the impact of state abstraction on an EFSM behaviour.

Proposition. Given a configuration $(s, \vec{v})$ of A and a factor EFSM A_π, each input-output sequence, defined in $(s, \vec{v})$ is also defined in $(\pi(s), \vec{v})$.

Therefore, the factor machine preserves behaviour of the original machine, while possibly adding a new behaviour.

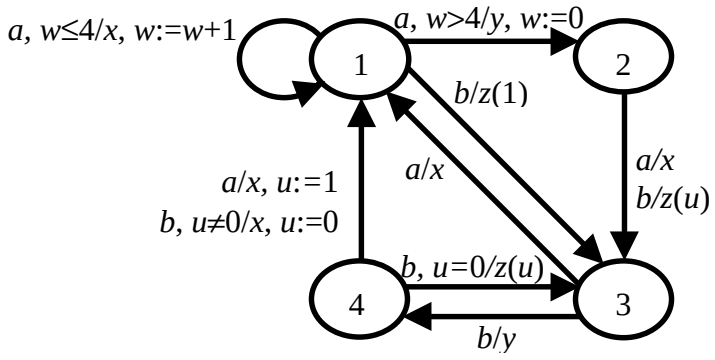

Figure 2: An EFSM.

Figure 3 represents an abstraction of this EFSM defined by the partition {{1, 3}, {2, 4}}. Another abstraction is given in Figure 4, it is defined by the partition {{1, 4}, {2, 3}}.

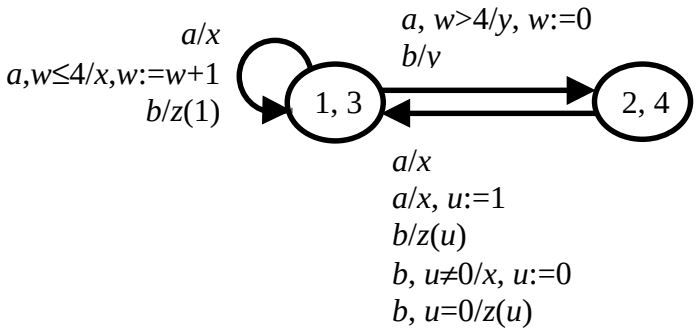

Figure 3: An abstracted EFSM.

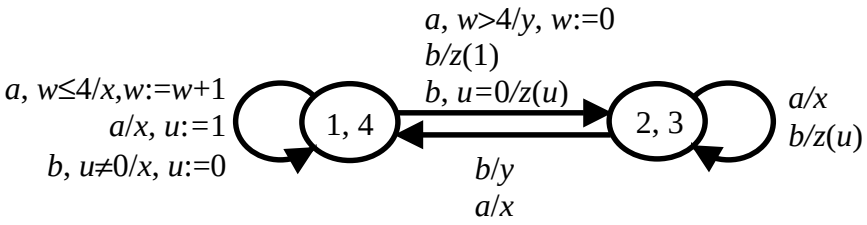

Figure 4: Another abstracted EFSM.

When states are merged, only the extremities of transition edges are modified. The definition requires that for each transition (s—$x,P/op,w,up$→s') of the original EFSM, the corresponding transition in the factor machine should have the same label $L = x,P/op,w,up$ associated to the edge. When two transitions s—L→s' and t—L→t' differ in the original EFSM only in their extremities, viz. $s \neq t$ and/or $s' \neq t'$, then if their extremities are merged viz. $\pi(s) = \pi(t)$ and $\pi(s') = \pi(t')$ the two transitions in the factor machine can be identified, and only one transition kept in this factor machine. If, however, $\pi(s) = \pi(t)$ but $\pi(s') \neq \pi(t')$ the factor EFSM could include the most tricky type of non-determinism, whereby it would not be observable [PBG99] even when the original EFSM is purely deterministic. A number of methods, especially for testing, apply only to observable machines [PBG99][BP+02][RBJ00] (the latter work deals with deterministic extended LTS which often could be viewed as observable EFSM). This problem has been addressed in an extension of our work which is reported in [Yun01] where a tool has been developed to perform observabilisation.

3.3 Discrepancies between EFSM and SDL

SDL processes have several structures that are absent in EFSM. In fact, syntax and semantic discrepancies between 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 analyse how these discrepancies affect state and transition merging. At the same time, we indicate some implementation choices of our tool, although we do not claim to 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.

Transitions. The correspondence between SDL transitions and transitions of EFSM is not direct. In SDL, a 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. We believe that the starting state, triggers, terminators, should be considered as components of a transition. Therefore, SDL transition may correspond to a set of EFSM transitions. We use the classical correspondence 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. 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 that are 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 can be determined by a static analysis of a process, it will not cause any problem for SDL transformation.

Non-determinism. While non-determinism in EFSM arises when it has several transitions sharing the same input, standard SDL processes may have non-determinism only in spontaneous transitions, actions and decisions. These are ANY decision and ANY expression. Therefore, in the ITU SDL, non-deterministic transitions could be modelled only with ANY decisions. However, we based our implementation on Object Geode which allows non-determinism in input clauses. ANY expressions are difficult to effectively represent in an EFSM model, but we do not see how this may affect state abstraction.

Time. Unlike our EFSM model, SDL processes are timed. Time, in fact, is considered as a designated variable, which may progress non-deterministically. Usually, in SDL processes, time aspects are expressed by a set of timers. Semantics of time progress and timers is complex. Therefore, we implement a pragmatic approximation of a timer by an input and a Boolean variable that indicates when the timer is set.

Procedures. Unlike EFSM, SDL transition may call procedures. This complicates the variable dependence analysis. In general, the procedures may contain their own states and transitions. There exist methods and tools for flattening SDL process. For example SDL2IF, available from www-verimag.imag.fr, translates SDL specifications into a procedure-free language IF [BF+99], which is often viewed as a flattened SDL with a sound time model. In our implementation, we had to assume that all procedures (especially with states) have been flattened manually or automatically, otherwise a conservative abstraction is not guaranteed.

Discard. To insure correctness of abstraction, implicit transitions should be taken into account. In our tool, we just assumed all inputs would be manually explicitly defined for each state when a conservative abstraction is needed.

Queue and related clauses. Explicit SAVEs, implicit saves (guarded transitions), and continuous signals do not breach the EFSM model. A queue could be modelled by a variable (a dynamic array); saves could be modelled with transitions, which modify this array; continuous signals could be modelled with a designated empty input signal of EFSM, that triggers such a transition. But that might not always be convenient. 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.

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

Other discrepancies between SDL and EFSM, would not affect SDL abstraction: these include exceptions (could be a special message), termination (sink states), alternatives, RPC.

3.4 State Abstraction (Merging) in 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 class of the partition could be merged into a single state of the resulting SDL process. Recall that states of the abstracted EFSM 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 since SDL state names are alphanumeric strings. Therefore, we need to introduce a correspondence from state sets into state names. We decided to use a simple but intuitive and clear mapping: concatenations of state names. 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 class in the partition. The order in which these state names are concatenated is not important, however it should be the same for each class. This is just a simple scheme to deal with "reasonable" specifications in our prototype. Of course, we could implement slightly more complex naming schemes to deal with potential name conflicts.

The proposed transformation results in a syntactically correct process, which is a conservative abstraction, provided that the original SDL process has no queue related operations, shortcuts, or implicit transitions (which could be unfolded prior to abstraction). The proposed state merging procedure is not conservative for SDL processes that use queue dependent transition clauses, as was explained above. Also, since SDL syntax forbids the use of several instances of the same input in the same state, a conservative state merging requires transition merging. If needed, an ANY decision may be used in a merged transition to model non-determinism. Figure 5 depicts from left to right a fragment of specification, a fragment of specification abstracted by a state name replacement (that is what our tool produces for Object Geode which allows multiple occurrences of an input under a state), and the same fragment transformed to meet the ITU SDL standard.

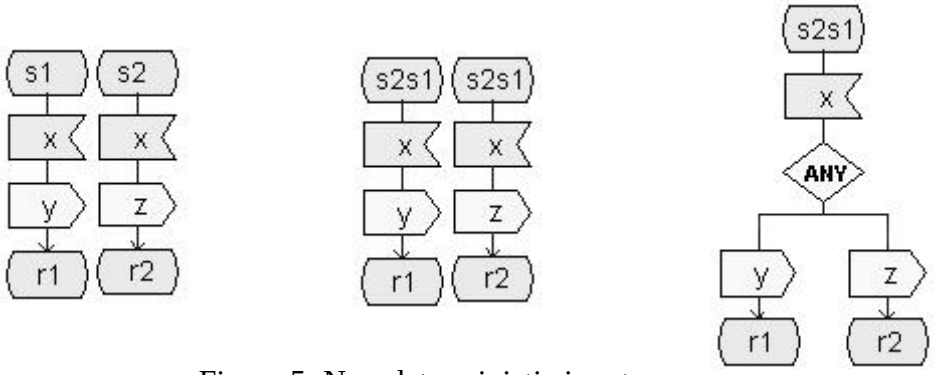

Figure 5: Non-deterministic inputs.

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 merging procedures.

Since the SAVE of a signal is equivalent to the input guarded with an identical FALSE enabling condition, we concentrate on the more general case of implicit saves (enabling conditions). A

possible solution is 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 6 (a). The state diagram (b) presents the result of state renaming (as implemented in our tool). The diagram (c) is a correct and conservative abstraction of these states. A refined diagram (d) of Figure 6 presents a finer, more precise, but at the same time, more complex abstraction of these states. However, what matters is not the complexity of the specification, but the number of states in the reachability graph.

Note that a simple disjunction of original predicates does not lead us to a conservative abstraction, because we must allow for potential refusals. For instance, we must preserve the behaviour of the specification where in state $s1$ it would not input x if $i \leq 1$ even though $j > 2$. Both diagrams (c) and (d) allow this refusal, and add new behaviours such as refusing x in $s2$ when $j > 2$.

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.

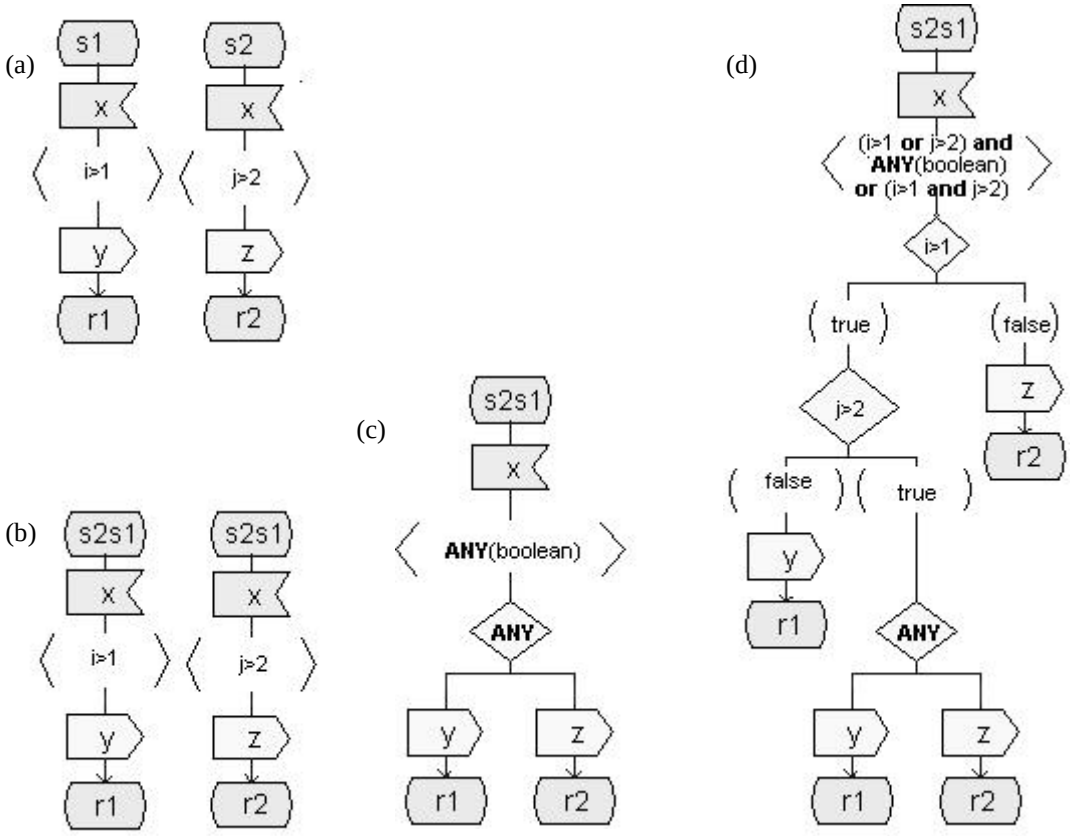

Figure 6: Merging states with enabling conditions.

3.5 Experimental State Abstraction Tool

The program merges states of a process according to a specified partition and transforms the process transitions in a form convenient for further processing (e.g., observabilisation). This tool is a result of the Master project of Hu Yun [Yun01]. It is written in Objective Caml ver. 3.00. Just as the variable abstraction tool, it uses the Object Geode SDL API to load an SDL specification textual file into a C data structure. An interface, developed at FranceTelecom R&D, transforms this C data

structure into an ML data structure and allows printing out this data into an SDL-like form (not strictly SDL, but sufficient for the rest of our processing).

The program receives the specification name as a command line argument. Then, the program outputs names of states of a process, and the user defines a partition of the state set. Then abstraction is performed and the result is written into an SDL output file that can be processed by other tools.

4. EXPERIMENTS

To assess the impact of our abstraction techniques, experiments on an SDL specification for ISDN telephone services proposed by another group in FranceTelecom R&D, have been performed. One of the concerns in using abstraction is to reduce the complexity of the specification and of its behaviour, esp. in the view of reachability analysis. Therefore, we experiment various abstractions with our tools, and compare the size of the graph and the coverage that could be achieved using the Object Geode verifier.

Of course, those are only limited experiments on a given specification with sample abstractions. We do not pretend that those experiments provide a thorough assessment of the proposed techniques. They just give a flavour of their impact on reachability analysis. The real upshot is that thanks to those abstraction techniques, we were able to implement test generation techniques as described in [BP+02].

4.1 An ISDN Telephone Service Case Study

The PR file specification for our case study contains about 2600 lines (about a hundred pages of GR). It describes a telephone service component, which is itself extracted from a more general architecture for telephone services. The specification contains one process and six procedures. It has seventeen states, four input signals, four output signals, thirty-four variables and three timers. Three input signals and one output signal are parameterised. The Object Geode verifier cannot completely explore such a complex specification. In one hour it explores about one million configurations delivering 100% state and 93% transition coverage of the specification, and during next seven hours it adds only about two hundred thousand configurations, whereas the depth of the simulation reaches nine.

The results of the specification simulation of a limited depth are shown at Table 1. Note that the transition coverage rate is determined in terms of the specification transitions and not the system reachability graph edges (which are unknown prior the simulation and can be infinite in the general case). Similarly, the state coverage is expressed in terms of the SDL states, not the nodes of the reachability graph (configurations). Those coverage measures simply relate to the degree of exercising the syntactic description of the specification, but the key impact of abstraction is on the parameters of the reachability graph..

Table 1: Simulation Results for the Specification.

Exploration depth	**5**	**6**
Number of configurations reached	887	5875
State coverage rate	79%	100%
Number of transitions covered	921	12411
Transition coverage rate	44%	63%

4.2 State Abstraction

In this experiment, we merge several states, reducing their number from seventeen to twelve. Table 2 shows simulation data for the obtained SDL specification. Comparing Table 1 and Table 2, we can see that, for the same exploration depth limitation, the Object Geode verifier performs much better on the abstracted SDL specification than on the original file. Merging states allows a significant increase in the exploration of the SDL system. This is no surprise of course since abstraction increases non-determinism. At a given depth, we can reach configurations through paths which are infeasible in the original specification. Better coverage of transitions is also achieved simply because more transitions are enabled in the abstracted version. Using this state abstraction is just an intermediate step towards a combined abstraction which better represents our fault model.

Table 2: Simulation Results for the Abstracted Specification.

Exploration depth	**5**	**6**
Number of configurations reached	5683	42579
State coverage rate	79%	100%
Number of transitions traversed	21319	210625
Transition coverage rate	62%	80%

4.3 Combining State and Variable Abstraction

Now, we further abstract the specification by projecting out twenty-seven variables from the obtained specification. The complete reachability graph exploration becomes feasible: in several seconds, 6294 states are reached and 32618 transitions are traversed with 100% coverage of the specification states and transitions. The depth of simulation is 61.

When we further abstract all the remaining variables, the depth of fifteen leads to complete simulation, resulting in 133 configurations and 636 transitions of the reachability graph. The result may look surprising since now the abstracted specification has only twelve states and no variables. The accounted configurations are intermediate nodes of the reachability graph, which have no counterpart in our EFSM model and are inserted by the Object Geode verifier mainly by technical reasons.

4.4 Discussions

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. Our SDL abstraction techniques can be used for automatic test generation, verification and other related activities.

However, there is a price to pay, esp. if we are interested in verification. The abstracted system has a richer behavior than the original system, so a 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.

Although the use of abstractions for verification purposes needs to be careful about preserving the semantics of the specification w.r.t. the properties to be verified, this was less relevant in our context where abstractions are mainly used to reduce the complexity of fault models, whose behaviours anyway extend to a superset of the behaviours of the original specification.

5. CONCLUSION

In this paper, we presented abstraction techniques for SDL processes. An abstraction is a simplified version of an SDL description. The simplification must be such that it preserves the behaviours of interest w.r.t. a given problem. For instance, in our case, abstraction was mainly used for a compact representation of fault models, i.e. an abstraction of a reference specification captures the behaviours of a class of faulty implementations of the reference. This in turn was useful for our technique of test generation. Abstraction could be also used for other problems, such as verification: for a given property to be verified, we can abstract from parts which are not relevant to the verification of that property.

So far, abstraction has mainly been used for verification, at least in the SDL community. Therefore, research work has mostly concentrated on abstraction techniques preserving semantic properties of the original specification. We consider that abstraction can be put to more uses, and this may require other types of abstractions.

Our abstraction techniques are generic, and do not depend on the type of problem considered (test generation, verification, synthesis of controllers, documenting, reverse engineering etc.). We considered that the variables or states of the SDL specification to be abstracted are a given input to our abstraction tools.

We adapted classical data-flow analysis techniques to abstract from designated variables and their dependencies. We proposed approaches to extend state abstraction as known in the FSM field to SDL specifications. Our algorithms were implemented on prototype tools that work on top of the Object Geode (parser) API. Those tools have been applied on typical applications from telecommunications.

We consider our work as a preliminary step on the use of abstraction on SDL specifications. This work has already shown its effectiveness in reducing drastically the complexity of our fault models, which was a necessary step to compute distinguishing test sequences in our approach [BP+02]. This work should be consolidated in the following directions.

First, we could attempt to redefine it more abstractedly in the semantic well-founded framework of abstract interpretation [Cou77]. At the same time, it could be extended to better combine state and variable abstraction with abstraction from other features of the language, in particular inter-process communication. Currently, our work has centred on dealing separately with variable abstraction and state abstraction in a single process. Finally, it would be worthwhile developing a compositional approach in a consistent toolbox for abstracting SDL specifications, preferably trying to stay within the limits of syntactically correct SDL; which has been one of our concerns, so as to be able to interwork with other SDL tools.

ACKNOWLEDGMENTS

This work was partly supported by France Telecom (research contract 001B341). The work of Yves-Marie Quemener was partly supported by the European Commission (IST project

ADVANCE, contract No IST-1999-29082). The work of Alex Petrenko was partly supported by the NSERC grant OGP0194381. Implementation of the algorithms on state and variable abstraction was mostly carried out by Hu Yun and Xiaoyu Wang as part of their MSc at Université de Montréal and McGill respectively; they also contributed to the experiments with our simplified case studies. We would like to thank Telelogic for the ObjectGeode license given to CRIM. We also thank our anonymous SAM reviewers for their helpful comments.

REFERENCES

[BD+91] v. Bochmann G., Das, A. Dssouli R., Dubuc M., Ghedamsi A., Luo G. "Fault Model in Testing." *Proceedings of the IFIP IV Workshop on Protocol Test Systems*; October 1991; Leidschendam, The Netherlands.

[BF+99] Bozga M., Fernandez J.-Cl., Ghirvu L., Graf S., Krimm J.-P., Mounier L., Sifakis J. "IF: An Intermediate Representation for SDL and its Applications." *Proceedings of SDL-FORUM 1999*; pp. 423–440, June 1999; Montreal, Canada.

[BFG99] Bozga M., Fernandez J.-Cl., Ghirvu L. "State Space Reduction based on Live Variables Analysis." *Proceeedings of SAS'99*, Venice, Italy. LNCS 1694, pp. 164–178, Springer Verlag.

[Boz99] Bozga M. *Vérification symbolique pour les protocoles de communication.* PhD Thesis, Grenoble University, 1999.

[BP+02] Boroday S., Petrenko A., Groz R., and Quemener Y.-M. "Test Generation for CEFSM Combining Specification and Fault Coverage." *Proceedings of IFIP 14th International Conference on Testing of Communicating Systems (TestCom 2002)*, March 2002, Berlin, Germany, Kluwer.

[CGP99] Clark E., Grumberg O., and Peled D. *Model Checking*. MIT, 1999.

[CLR92] Cormen T., Leiserson C., Rivest R. *Introduction to Algorithms*. MIT, 1992.

[Cou77] Cousot P. and Cousot R., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." *4th POPL*, pp. 238–252, Los Angeles, USA, 1977, ACM Press.

[Gil62] Gill A., *Introduction to the Theory of Finite-State Machines*, McGrawHill, New York, 1962.

[GS+96] Grabowski J., Scheurer R., Toggweiler D., 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.

[KJG99] Kerbrat A., Jéron T., Groz R. "Automated test generation from SDL specifications." *Proceedings of the 9th SDL Forum*; June 1999; Montreal, Canada. Elsevier.

[LG+95] Loiseaux C., Graf S., Sifakis J., Bouajjani A., and Bensalem S., "Property Preserving Abstractions for the Verification of Concurrent Systems.", *Formal Methods in System Design*, 6, pp. 11–44, 1995.

[MAH98] Moundanos D, Abraham J.A., Hoskote Y., "Abtraction Techniques for Validation, Coverage Analysis and Test generation." *IEEE Trans. on Computing*, Vol. 47, No. 1, Jan. 1998, pp. 2–14.

[Oik96] Oikonomou K.N., "Abstractions of Finite-state Machines and Optimality with Respect to Immediately-Detectable Next-State Faults.", *IEEE Transactions on Systems, Man, and Cybernetics*, Part A, pp. 151–160, Vol. 26, No.1, January 1996.

[Pet00] Petrenko A. “Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography.” LNCS 2067, Proceedings of the Summer School MOVEP'2000, *Modeling and Verification of Parallel Processes*; June 2000; Nantes, France.

[PBG99] Petrenko A., Boroday S., and Groz R. “Confirming configurations in EFSM.” *Proceedings of the IFIP Joint International Conference on Formal Description Techniques for Distributed Systems (FORTE XII) and Communication Protocols, and Protocol Specification, Testing, and Verification (PSTV XIX)*; October 1999, China. Kluwer.

[PeY92] Petrenko A. and Yevtushenko N. “Test Suite Generation for a FSM with a Given Type of Implementation Errors.” *Proceedings of the IFIP 12th International Symposium on Protocol Specification, Testing, and Verification*; 1992; USA. North-Holland.

[RBJ00] Rusu V., du Bousquet L., and Jeron T. “An Approach to Symbolic Test Generation.” *International Conference on Integrating Formal Methods (IFM'00)*, Springer Verlag, LNCS 1945, pp. 338–357, Novembre 2000.

[Tip95] Tip F. “A survey of program slicing techniques.” *Journal of Programming Languages*, 3(3), pp. 121–189, September 1995.

[Wan01] Wang X. Abstraction of Variables in SDL. Master Thesis, McGill University, 2001.

[Yun01] Yun H. State abstraction in SDL. Master Thesis, Université de Montréal, 2001.