Extending the Translation from SDL to Promela

Armelle Prigent[1], Franck Cassez[2], Philippe Dhaussy[1] and Olivier Roux[2]

[1] ENSIETA, Brest, France <firstname.name>@ensieta.fr
[2] IRCCyN, ECN, Nantes, France <firstname.name>@irccyn.ec-nantes.fr

Abstract. In this paper, we tackle the problem of model-checking SDL programs that use the `save` operator. Previous work on model-checking SDL programs with SPIN consists in translating SDL into IF (using `sdl2if`) and finally IF to Promela (`if2pml`). Nevertheless the `save` operator of SDL is not handled by the (final) translator `if2pml`. We propose an extension of the tool `if2pml` that translates IF programs with `save` operators into Promela. We also add an abstraction method on buffers messages to `if2pml` allowing the user to gather some buffer messages into one abstract value.
We use our extended version of `if2pml` to validate an Unmanned Underwater Vehicle (UUV) subsystem specified with SDL.

Keywords: SDL formalism, save operator, model-checking, data abstraction

1 Introduction

SDL for industrial applications. The developments of embedded reactive systems are subject to a tight integration of the formal methodologies into the existing software development cycle in order to increase the quality of the design. Our research group is involved in the design of advanced robotics control systems and we have recently developed pieces of software for an Unmanned Underwater Vehicle (UUV). In this project, and due to various industrial requirements, we had to specify the system with the SDL formalism, normalized by ITU[1], recommendation Z.100 [ITU94b]. The software has many critical parts involving (ad hoc) communication protocols we have developed hence the need for a formal verification of safety requirements.

SDL and formal verification with SPIN. The SDL tools VERILOG [VER99] and TELELOGIC [TEL98] allow the user to check for a restricted subset of properties like deadlocks, infinite loops or exceeded queue lengths. In many cases those safety requirements are not sufficient to ensure a good software quality and the need for expressing more subtle properties (e.g. using temporal logics) arises. In order to check temporal properties on SDL specifications, Bosnaki & al. [BDHS00] have proposed to translate SDL specifications into Promela programs that can be model-checked with SPIN [Hol97]. The technic

[1] International Telecommunication Union

consists in (1) translating a SDL program into the intermediate format IF (via `sdl2if` [BFG+99]) ; (2) the IF program is then translated into Promela (using `if2pml` presented in [BDHS00]).

Our contribution. In the development of the UUV system, we make an extensive use of the SDL `save` operator. Although this operator exists in the IF language the translation of the IF `save` operator into Promela with the tool `if2pml` has not been implemented yet. We then had to extend the tool `if2pml` to handle this operator. One of the impediment we encountered in the development and model-checking of our UUV software was of course the state-explosion problem. This problem was even amplified because of the translation of the `save` operator that duplicates buffers and so brings about an exponential growth in the number of states of the system. To tackle this problem we have again extended the tool `if2pml` with a *message abstraction* capability so that some messages can be gathered and abstracted away following the method proposed by Clarke & al. in [CGL92].

Outline of the paper. The paper is organized as follows: section 2, deals with the implementation of the translation of the `save` operator into `if2pml` to produce Promela programs. The next section 3 is devoted to the presentation of message abstraction via Clarke's abstraction algorithm [CGL92] and its implementation in an extended version of `if2pml`. Finally, the application of the above technics are presented on the UUV system in section 4 and we conclude in section 5.

2 Translating SDL Events' Savings into Promela

2.1 SDL programs

An SDL program consists of a set of processes described in a graphical language. Each process has an *input FIFO queue* in which events to be processed are stored. A process can output events to other input queues. The informal semantics of a single step of an SDL process[2] is roughly:

1. process an event from the input queue;
2. output events,
3. go to step 1.

The communication between processes is asynchronous. One of the features of SDL processes is the capability of *storing* events in order to process them later. This capability is very similar to the one used in the Electre reactive language [CR95] where the semantic model is a FIFFO (First In First Fireable Out) automaton [SFRC99]. In SDL programs, some of the events of the input queue cannot be processed in particular states and are then stored for later processing. This feature is explicitly implemented with the SDL `save` operator. An event in

[2] see [ITU94a] for a formal definition.

a queue is actually a complex structure which contains the SDL identifier of the events (its name), (the list of SDL data values carried by the event) and the Pid of the sender: in the following we will only deal with the event and the Pid value attached to it (e.g. $c(sender)$ for event c that was sent by process $sender$).

2.2 The save operator

A `save` operator specifies a set of events that cannot be processed in a particular state and are to be kept in the input queue for later processing. Figure 1 gives an example of the use of the (SDL graphical) `save` operator. When the process is in state $wait$ only events c, f can be processed, whereas a, b must be left in the input queue, and d, e are neither saved nor processed and thus are discarded[3]. Based on [ITU94b], the formal semantics for the processing of events for one process P is the following:

- let $\mathcal{B}_s$ be the set of saved events when process P is in state s, $\mathcal{T}_s$ be the set of events that can be processed in state s, $\widetilde{\mathcal{T}}_s$ the set of events that are discarded in state s; then $\mathcal{B}_s \cup \mathcal{T}_s \cup \widetilde{\mathcal{T}}_s = \mathcal{E}$ is a partition of the set of all input events of P;
- let $\varphi \in \mathcal{E}^*$ be the current input queue when process P is in state s,
- then
  - either $\varphi = w.e.w'$ with $e \in \mathcal{T}_s$, $w \in (\mathcal{B}_s \cup \widetilde{\mathcal{T}}_s)^*, w' \in \mathcal{E}^*$. From state s, P will reach a new state s' and the new queue is[4] $\varphi' = w_{|\mathcal{B}_s}.w'$.
  - or $\varphi \in (\mathcal{B}_s \cup \widetilde{\mathcal{T}}_s)^*$. in this case no event can be processed and the queue is left unchanged: $\varphi' = \varphi$. This agrees with the semantics of discarded events given in [BDHS00] (which is different from the one in [ITU94b]).

To sum up, P will process the first (the oldest) *non saved* event of its input queue if it can be taken into account in state s. All the preceeding discarded events are removed from the queue.

For instance, if $\mathcal{E} = \{a, b, c, d, e\}$, in state $wait$ of Figure 1 we have $\mathcal{B}_s = \{a, b\}$, $\mathcal{T}_s = \{c, f\}$ and $\widetilde{\mathcal{T}}_s = \{d, e\}$. If the input queue is $abcd$, c will be processed and the new input queue is abd. Now if the input queue is $abdc$, c is processed and d removed leading to ab. If the queue is $abddcfd$, a single step will lead to $abfd$.

2.3 Translation of SDL save into IF save

An SDL process P is translated into an IF process `proc_P` by the `sdl2if` program presented in [BFG+99]. In this translation, an input buffer `q_proc_P` is associated to `proc_P`. Figure 2 gives the IF code of the sample part of the process depicted on Figure 1. The translation of the IF process into Promela does not yet take into account the saved events.

[3] actually, d(or e) will be discarded if it is before the first processable event in the queue.

[4] for $\Sigma' \subseteq \Sigma$ and $w \in \Sigma^*$, we denote by $w_{|\Sigma'}$ the word obtained from w by removing all the letters not belonging to Σ'.

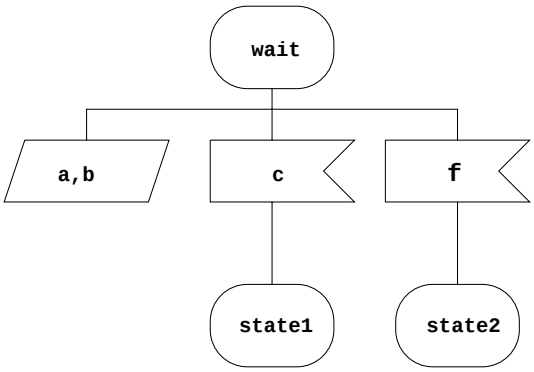

Fig. 1. Sample part of an SDL process using saved events

```
process proc_i0 :buffer q_proc_i0;
  var
    sender : pid;
    parent : pid;
    offspring : pid;
  state
    start :init;
    wait
      discard d,e
        in q_proc_i0;
      save a,b
        in q_proc_i0;
      end;
    state1;
    state2;
  transition
    from start to wait;
    from wait
         input c(sender) from q_proc_i0 to state1;
    from wait
        input f(sender) from q_proc_i0 to state2;
    from state1 to wait;
    from state2 to wait;
```

Fig. 2. The translated IF code for the SDL state *wait*

Nevertheless the semantics of the save operator exists in the IF language. The dynamic semantics of IF programs [BGG+99] that gives the meaning of the save IF construct is essentially the same as the one we have given in section 2.2. The translation of an SDL program composed of n processes is an IF program composed of the n IF translations of the processes.

The crucial points when implementing this semantics in Promela is that it implies a recursive processing of the input queue until an event that can be processed is found. This type of search and dequeing anywhere in the queue cannot be translated directly using Promela primitives.

2.4 Translation of IF save into Promela

A naive way of translating the processing of a queue φ in state s into Promela that preserves the semantics of the save operator would be:

- let $\mathcal{B}_s$ be the set of saved events in s, $\widetilde{\mathcal{T}}_s$ the set of discarded events and $\mathcal{T}_s$ the set of processable events,
- process φ as follows:
  1. if $\exists e \in \varphi \wedge e \in \mathcal{T}_s$ then
     (a) add a fresh end token $\perp$ at the end of the queue φ,
     (b) do :
     dequeue e' from φ and if $e' \in \mathcal{B}_s$ enqueue e' in φ
     until $\varphi = e.w$ with $e \in \mathcal{T}_s$;
     (c) remove e from φ and change the state of the process P according to the e-transition;
     (d) dequeue e' from φ and enqueue e' while $e' \neq \perp$;
     (e) dequeue $\perp$.
  2. otherwise do nothing.

It is quite obvious that for a buffer of length n every processing needs at most $2(n+1)$ steps of dequeuing + inqueuing. In practice, this algorithm requires a free slot to enqueue $\perp$. This would mean that if we want to store at most n events, we take an actual buffer of length $n+1$. Moreover before any enqueuing we have to add a test in the Promela program on the length of the buffer.

We present a solution that uses a temporary queue but avoids testing buffer length at each enqueuing. Suppose the queue is of the form $\phi = w.e.w'$ with $e \in \mathcal{T}_s$, $w \in (\mathcal{B}_s \cup \widetilde{\mathcal{T}}_s)^*, w' \in \mathcal{E}^*$ (with $\mathcal{B}_s, \mathcal{T}_s, \widetilde{\mathcal{T}}_s$ defined in section 2.2).

The algorithm using an intermediate queue φ' consists in:

1. do :
   dequeue e' from φ and if $e' \in \mathcal{B}_s$ enqueue e' in φ'
   until $\varphi = e.w$ with $e \in \mathcal{T}_s$;
2. process e: remove e from φ and change the state of the process P according to the e-transition;
3. do :
   dequeue e' from φ and enqueue e' in φ'
   until $\varphi = \epsilon$ (ϵ is the empty word);

φ' contains the new updated queue.

For a buffer of length n every processing needs at most $2(n+1)$ steps of dequeuing + inqueuing. About space complexity, we use another intermediate buffer φ'. As $|\varphi'| = |\varphi| = n$, we need more space that the naive algorithm presented below.

We illustrate our algorithm on Figure 3.

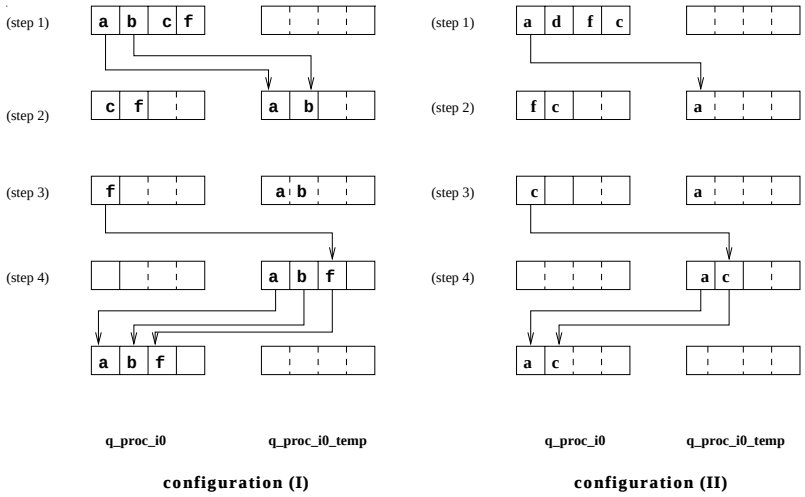

Fig. 3. Reception principle with two configurations

Let `wait` be the SDL state mentionned in Figure 1. The processing of events c or f is considered with two configurations (I) and (II). The left-hand buffer is the input buffer of the process (φ) and the right-hand one the temporary buffer φ'. From step 1, we enqueue saved events leading to step 2. In step 2, the first event of φ is a non saved event and is processed. Step 3 consists in enqueuing each event in φ' until φ is empty. In the last step, each event temporaly stored in φ' is appended to φ ($[\varphi := \varphi']$). This step respect the initial event order in the buffer φ. The translation of the IF program of Figure 2 into a Promela program is given on Figure 4. The process `proc` manages two buffers: the initial one `q_proc_i0` corresponds to φ and the temporary buffer `q_proc_i0_tmp` is φ'. They are declared with the same size as the initial SDL buffer. The first test (step 1) is to verify that one of the waited events (c or f) is in the buffer (`q_proc_i0??[c] || q_proc_i0??[f]`). The saved events preceeding c or f in the buffer are then stored in the temporary buffer (step 2) (`q_proc_i0_tmp!a` and `q_proc_i0_tmp!b`). Discarded events are consumed and removed from the queue. When c or f is encountered (`q_proc_i0?c or q_proc_i0?f`), a first loop is executed. This one consists in appending all events still in the initial buffer to the temporary buffer. When `q_proc_i0` is empty, the last step is to move `q_proc_i0_tmp` into this buffer. Each event is appended to the buffer until the

emptyness of the temporary buffer `q_proc_i0_tmp`. When the new queue is ready in buffer `q_proc_i0`, the actions associated with the processed event are taken and the new state is reached.

Our extended version of the `if2pml` translator implements this algorithm and the Promela code of Figure 4 is an example of its output.
The implementation of the `save` primitive requires a new temporary queue for each input queue. Then the number of states of the resulting Promela program is multiplied in the worst case by the number of states of the FIFO queues.

Of course this does not rule out situations where the queue is full and one has to make sure that the length of the queue is large enough to handle all the pending events.

The systems we are developing make use of a lot of buffers and the state blow up is particularly high when considering the number of events in a buffer. Let B be a buffer with p places in which k event values can be stored. This buffer has $\sum_{n=0}^{n=p} k^n$ possible values. So, a 3-place buffer with 4 different events has 85 possible values.

To overcome this problem we give in the next section an implementation of an abstraction method on buffer messages.

3 Message buffer abstraction

3.1 Abstract interpretation

Abstract interpretation [CC77] consists in building an abstract model $\widehat{M}$ of a system from a concrete one M preserving some relations between the two models. The aim is to reduce the state space of the system such that some properties of the system are preserved from the abstract to the concrete model.

Usually the abstract system constructed has more behaviors than the initial program, and the *preservation result* [CGL92] states that properties quantifying over all paths of the abstract system are preserved whereas existentially quantified properties are not.

For instance the preserving result applies to a subset of the branching time logic CTL^* in which only the path quantifier $\forall$ is allowed: this subset is usually referred to as $\forall CTL^*$. As a consequence it can also be applied to LTL properties quantifying over all paths[5]. Of course the formula on the abstract model has to be expressed in term of the abstract data: let $\widehat{\phi}$ denote the abstract property obtained from the concrete property ϕ[6]. Then, for a formula $\phi \in \forall CTL^*$, if $\widehat{M} \models \widehat{\phi}$ then $M \models \phi$.

3.2 Data abstraction

The abstraction algorithm of [CGL92] consists in interpreting the concrete program to obtain directly the abstract version of the system. The initial model

[5] as LTL is a subset of $\forall CTL^*$.

[6] $\widehat{\phi}$ depends on the abstraction mapping chosen to build $\widehat{M}$.

q_proc_i0??[c]|| q_proc_i0??[f] allows the processing only if one of the waited events is in the input queue

```
wait:
atomic{
if
::(q_proc_i0??[c]|| q_proc_i0??[f])->
   do
   :: q_proc_i0?b(sender)->q_proc_i0_temp!b(sender);
   :: q_proc_i0?a(sender)-> q_proc_i0_temp!a(sender);
   :: q_proc_i0?e,_->
   :: q_proc_i0?d,_->
   :: q_proc_i0?c->
```

When event c is at the head, the first loop consist in dequeue each event until the emptiness of the input buffer

```
        do
        :: q_proc_i0?b(sender)-> q_proc_i0_temp!b(sender);
        :: q_proc_i0?a(sender)-> q_proc_i0_temp!a(sender);
        :: q_proc_i0?e(sender)-> q_proc_i0_temp!e(sender);
        :: q_proc_i0?d(sender)-> q_proc_i0_temp!d(sender);
        :: q_proc_i0?c(sender)-> q_proc_i0_temp!c(sender);
        :: q_proc_i0?f(sender)-> q_proc_i0_temp!f(sender)
        :: empty(q_proc_i0)-> break;
        od;
```

When q_proc_i0 is empty, the second loop replaces all events in the initial order in this buffer while dequeuing the temporary buffer

```
        do
        :: q_proc_i0_temp?b(sender)-> q_proc_i0!b(sender);
        :: q_proc_i0_temp?a(sender)-> q_proc_i0!a(sender);
        :: q_proc_i0_temp?c(sender)-> q_proc_i0!c(sender);
        :: q_proc_i0_temp?e(sender)-> q_proc_i0!e(sender);
        :: q_proc_i0_temp?d(sender)-> q_proc_i0!d(sender);
        :: q_proc_i0_temp?f(sender)-> q_proc_i0!f(sender)
        :: empty(q_proc_i0_temp)-> break;
        od;
        goto state1;

   :: q_proc_i0?f->
        do
        :: q_proc_i0?b(sender)-> q_proc_i0_temp!b(sender);
        :: q_proc_i0?a(sender)-> q_proc_i0_temp!a(sender);
        :: q_proc_i0?e(sender)-> q_proc_i0_temp!a(sender);
        :: q_proc_i0?d(sender)-> q_proc_i0_temp!a(sender);
        :: q_proc_i0?c(sender)-> q_proc_i0_temp!c(sender);
        :: q_proc_i0?f(sender)-> q_proc_i0_temp!f(sender)
        :: empty(q_proc_i0)-> break;
        od;

        do
        :: q_proc_i0_temp?b(sender)-> q_proc_i0!b(sender);
        :: q_proc_i0_temp?a(sender)-> q_proc_i0!a(sender);
        :: q_proc_i0_temp?e(sender)-> q_proc_i0!e(sender);
        :: q_proc_i0_temp?d(sender)-> q_proc_i0!d(sender);
        :: q_proc_i0_temp?c(sender)-> q_proc_i0!c(sender);
        :: q_proc_i0_temp?f(sender)->q_proc_i0!f(sender)
        :: empty(q_proc_i0_temp)-> break;
        od;
        goto state2;
   od;
fi;
```

Fig. 4. Promela code obtained from the SDL program of Figure 1

is a labelled transition system. This abstraction can be applied to IF programs during the translation into Promela. Indeed, each IF process is associated with a labelled transition system. The benefit of this method is that the abstract model is constructed directly from the initial program. This is particularly interesting for infinite or large systems.

In the sequel, we use Clarke's algorithm [CGL92] to build an abstract model of the system where some buffer messages are abstracted away. We apply this technic to the IF program obtained from a SDL program which is the concrete model, to build an abstract Promela version of the program.

3.3 Buffer abstraction on IF program

The abstraction algorithm we use is the algorithm of [CGL92], where the abstraction mapping deals with the buffer contents.

Let $\mathcal{E}$, represent the possible events that can be stored in a buffer B of the system. $\mathcal{E}^*$ is then the set of possible values of the buffer. We denote $\mathcal{E}^A$ the set of abstract events ($(\mathcal{E}^A)^*$ is then the set of abstract contents of the buffer.) The abstraction mapping $h : \mathcal{E} \to \mathcal{E}^A$ associates to each event $e_1, e_2, \ldots, e_n \in \mathcal{E}$ an abstract value in $\mathcal{E}^A$. We denote $h(e) = e^A$.

Together with the abstraction mapping, we have to define *abstract primitives* on buffers. The abstract operators on buffers are defined straightforwardly from the concrete by:

$$\begin{aligned} h(input(sig)) &= input(sig^A) \\ h(output(sig)) &= output(sig^A) \\ h(save(sig)) &= save(sig^A) \end{aligned}$$

The translation from IF to Promela is computed compositionaly. Indeed for a composition $(P_1|\cdots|P_n)$ of n IF processes with have $(P_1|\cdots|P_n)^A = (P_1^A|\cdots|P_n^A)$ where P^A denotes the abstract process obtained from P. The abstract interpretation then consists in constructing an abstraction for each IF process in the system and compose the abstracted processes.

Practically the abstract interpretation of the system is done during the translation of the IF system into the corresponding Promela program (`if2pml`).

3.4 Application

To illustrate this method, we apply the buffer abstraction on the state *wait* described in Figure 1. The buffer abstraction mapping gathers events `a` and `b` under the label `SIG_ABST`. Formally, using the relation h, we write : $h(b) = h(a) = SIG\_ABST$ and $h(x) = x$ for the other events. The abstracted version in Promela of the IF program of Figure 4 is shown Figure 5.

In state *wait*, the events a, b have been abstracted into SIG_ABST. This abstraction reduces the number of `save` in the transition relation and then the number of possible transitions in the system.

```
wait:
atomic{
if
::(q_proc_i0??[c]|| q_proc_i0??[f])->
   do
   :: q_proc_i0?SIG_ABST(sender)-> q_proc_i0_temp!SIG_ABST(sender);
   :: q_proc_i0?e,_->
   :: q_proc_i0?d,_->

   :: q_proc_i0?c->
        do
        :: q_proc_i0?SIG_ABST(sender)-> q_proc_i0_temp!SIG_ABST(sender);
        :: q_proc_i0?e(sender)-> q_proc_i0_temp!e(sender)
        :: q_proc_i0?d(sender)-> q_proc_i0_temp!d(sender)
        :: q_proc_i0?c(sender)-> q_proc_i0_temp!c(sender)
        :: q_proc_i0?f(sender)-> q_proc_i0_temp!f(sender)
        :: empty(q_proc_i0)-> break;
        od;

        do
        :: q_proc_i0_temp?SIG_ABST(sender)-> q_proc_i0!SIG_ABST(sender);
        :: q_proc_i0_temp?e(sender)-> q_proc_i0!e(sender)
        :: q_proc_i0_temp?d(sender)-> q_proc_i0!d(sender)
        :: q_proc_i0_temp?c(sender)-> q_proc_i0!c(sender)
        :: q_proc_i0_temp?f(sender)-> q_proc_i0!f(sender)
        :: empty(q_proc_i0_temp)-> break;
        od;
        goto state1;

   :: q_proc_i0?f->
        do
        :: q_proc_i0?SIG_ABST(sender)-> q_proc_i0_temp!SIG_ABST(sender);
        :: q_proc_i0?e(sender)-> q_proc_i0_temp!e(sender)
        :: q_proc_i0?d(sender)-> q_proc_i0_temp!d(sender)
        :: q_proc_i0?c(sender)-> q_proc_i0_temp!c(sender)
        :: q_proc_i0?f(sender)-> q_proc_i0_temp!f(sender)
        :: empty(q_proc_i0)-> break;
        od;

        do
        :: q_proc_i0_temp?SIG_ABST(sender)-> q_proc_i0!SIG_ABST(sender);
        :: q_proc_i0_temp?e(sender)-> q_proc_i0!e(sender)
        :: q_proc_i0_temp?d(sender)-> q_proc_i0!d(sender)
        :: q_proc_i0_temp?c(sender)-> q_proc_i0!c(sender)
        :: q_proc_i0_temp?f(sender)->q_proc_i0!f(sender)
        :: empty(q_proc_i0_temp)-> break;
        od;
        goto state2;
   od;
fi;
```

Fig. 5. Abstracted Promela code

3.5 An extension of `if2pml` to automatically abstract buffers

We have implemented the buffer abstraction for SDL programs via the the translation into IF. We have extended the translator `if2pml` developed by [BDHS00] with the abstraction feature. Our implementation uses a file describing the abstract mapping. For the example of Figure 5, this file (`fic.grp`) contains the following line: `SIG_ABST : a,b;` meaning that a and b are abstracted into the same event `SIG_ABST`. For the other event, abstraction is the identity mapping. It may be possible to have more than one abstract signal. Then using the command line `if2pml -a fic.grp prog.if` produces the given abstracted Promela code.

4 Case Study: Verification of the Obstacle Avoidance System of the UUV

4.1 Obstacle Avoidance System

The information system of UUV is based on a distributed architecture that comprises several subsystems. One is the Obstacle Avoidance System (OAS). The principle of this system we develop is to manage in an integrated way a digital terrain model estimation method, a 3D stabilized and mechanically steered front looking sonar, and computational methods devoted to safe trajectories computation. It is composed of four subsystems (inside the dashed box in Figure 6):

1. The digital terrain manager (DTM), which estimates the partially known terrain by using an occupancy grid representation and updating process,
2. the global planner (GP) to generate way points guiding the UUV towards a given target whilst avoiding terrain obstacles,
3. the reflex planner (RP) to check that the trajectories planned by the GP are safe, even in the presence of a disturbance, in the sense that they do not lead to collision,
4. the OAS Supervisor that manages the communication with environmental subsystems.

The communication between the OAS system and the other UUV systems is based on an Ethernet network and a CAN bus coupled to actuators and sensors. The internal OAS processes communication mechanism is via VxWorks message queues.

4.2 SDL model

The Obstacle Avoidance System has the three following operation modes: *Rerouting*, *Terrain following* and *Security*. When necessary, the supervisor is alerted by Mission Control (event `changeMode`) that a mode change is required. Periodically the Navigation System sends navigation data, Mission Control sends a target set to reach and the current target number (`nav, consign, wayPoint`

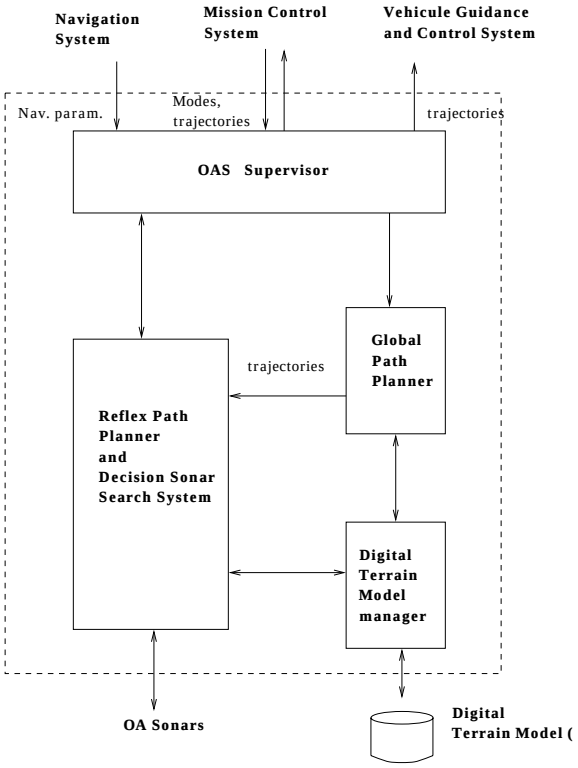

Fig. 6. Functional description of the Obstacle Avoidance System.

and `noWp`) to the supervisor process. This one will be in charge to redistribute it to other processes. The trajectory request comes from Control System with event `simuTimeout`. The trajectory is computed by the process concerned with the current operation mode (*Rerouting* or *Terrain following* or *Security*). The verification with the model-checker SPIN requires to close the system and we have defined an environment process model in SDL. This particular process is in charge of simulating interactions between the OAS SDL model and the Navigation System, Control System, Sonar and Mission Control System. Figure 7 presents the SDL system and events exchanges between these processes.

4.3 Verification of the OAS

Properties of the OAS. Using the tools described in section 2 and 3 we can check for different safety properties (of $\forall CTL^*$) of the OAS system. The properties we want to check are the following:

– *"the system does not get stuck in one of the operation mode"*; this means that the operation mode is alternatively changed during the execution of the system. The mode could be *terrain following* (`terfolMode`), *rerouting*

(`reroutMode`), or *security* (`securMode`). This property can be expressed in LTL by:

$$\neg\Diamond\Box\texttt{terfolMode} \wedge \neg\Diamond\Box\texttt{reroutMode} \wedge \neg\Diamond\Box\texttt{securMode} \quad (1)$$

– *"The trajectory is computed by the process concerned by the current operation mode"*. e.g. in *rerouting* mode, the trajectory has to be computed by process `rerout`. The trajectory is computed by a process when it receives event `nav1`. We then define `trajRerout` to be equivalent to process `rerout` receives event `nav1`: `q_rerouting_i0?[nav1]`. The LTL property to be checked is:

$$\Box\neg(\texttt{terfolMode} \wedge \texttt{trajRerout}) \quad (2)$$

– *"In a rerouting or security operation mode the* `sonar manager` *does not send data to the* `model` *process"*.

$$\Box\neg((\texttt{reroutMode} \vee \texttt{securMode}) \wedge \texttt{sentDataModel}) \quad (3)$$

All these properties involve a number of events in many buffers and the SDL description of the OAS system makes extensive use of `save` operators. We have used our extended version of `if2pml` to check for properties (1)–(3).

Results. Table 1 presents the different reduction percentages obtained on the OAS system for different abstraction mappings h_1 and h_2. h_1 groups 4 events into one, h_2 makes two groups of 4 events for each one. $h_0 = Id$ is the identity mapping giving the number of states and transitions of the concrete system. The ratio columns corresponds respectively to $1 - \frac{\#\ states\ of\ abstract}{\#\ states\ of\ concrete}$ and $1 - \frac{\#\ trans.\ of\ abstract}{\#\ trans.\ of\ concrete}$

	# states	# trans.	% states ratio	% trans. ratio
$h_0 = Id$	95 633	505 341	-	-
h_1	86 225	403 077	10 %	9 %
h_2	67 951	369 035	40 %	36 %

Table 1. Reduction of the number of states and transitions

Properties (1) and (3) are true on our model. Property (2) is violated. Indeed, the `supervisor` can receive a trajectory computed by the `terfol` process whereas *rerouting* mode is activated. As the relation between the abstracted and the initial model is a simulation, the property violation detected in the abstract system does not allow us to affirm the violation in the concrete model. Nevertheless, the MSC[7] of the incorrect behavior produced by SPIN has been analyzed

[7] Message Sequence Charts

and input to the SDL system. This incorrect behavior has been reproduced in the initial SDL system with a simulation. We have then checked that the property is really violated in the concrete SDL model. This incorrect behavior has been detected and fixed in our OAS model.

5 Conclusion and future work

In this paper we have extended the tool `if2pml` with the two following features:

- translation of the `save` operator into Promela;
- implementation of an abstraction mapping on buffers' messages.

These technics reveal useful when proving SDL programs that use the `save` operator. The designer can explicitly group messages into one abstract message. The reduction obtained in the number of states and transitions of the system we want to check are rather significant, going up to 40%. We have successfully applied our extended version of `if2pml` on the development of an Unmanned Underwater Vehicle for which safety properties were checked.

As an incorrect behavior was detected on the abstract system with SPIN for property (2), we had to verify that this incorrect behaviour really exists on the concrete initial SDL system. Such a verification could be automated using the SPIN MSC counter example to produce automatically the corresponding SDL MSC that could be played on the concrete SDL model.

References

[BDHS00] Dragan Bosnacki, Dennis Dams, Lesek Holenderski, and Natalia Sidorova. Model checking sdl with spin. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems,*, number 1785, pages 363–377, Berlin, 2000. LNCS, Springer.

[BFG+99] M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis. If: An Intermediate Representation for SDL and its Applications. In *Proceedings of SDL-FORUM'99, Montreal, Canada*, June 1999.

[BGG+99] M. Bozga, L. Ghirvu, S. Graf, L. Mounier, and J. Sifakis. The Intermediate Representation IF: Syntax and semantics. Technical report, Vérimag, Grenoble, 1999.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by cons truction or approximation of fixpoints. In ACM Press, editor, *Proceedings of the 4th Annual Symposium on Principles of Programming Languages,*, 1977.

[CGL92] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Proceedings of the 19th ACM symposium on principles of programming languages*, ACM press, New-York, 1992.

[CR95] Franck Cassez and Olivier Roux. Compilation of the ELECTRE reactive language into finite transition systems. *Theoretical Computer Science*, 146(1–2):109–143, July 1995.

[Hol97] G.J. Holzmann. The model checker spin. In *IEEE Trans. on Software Engineering*, volume 23, May 1997.

[ITU94a] ITU-T International Telecommunication Union. *Annex F.3 to Recommendation Z.100, Specification and Description Language (SDL) – SDL Formal Definition: Dynamic Semantics*. 1994.

[ITU94b] ITU-T International Telecommunication Union. *Recommendation Z.100, Specification and Description Language (SDL)*. 1994.

[SFRC99] G. Sutre, A. Finkel, O. Roux, and F. Cassez. Effective recognizability and model checking of reactive fiffo automata. In *Proc. 7th Int. Conf. Algebraic Methodology and Software Technology (AMAST'98), Amazonia, Brazil, Jan. 1999*, volume 1548 of *Lecture Notes in Computer Science*, pages 106–123. Springer, 1999.

[TEL98] TELELOGIC. *TAU/SDT 3.3*. TELELOGIC, June 1998.

[VER99] VERILOG. *ObjectGEODE 4.0*. CS VERILOG, March 1999.

A SDL Model of OAS

A.1 SDL System

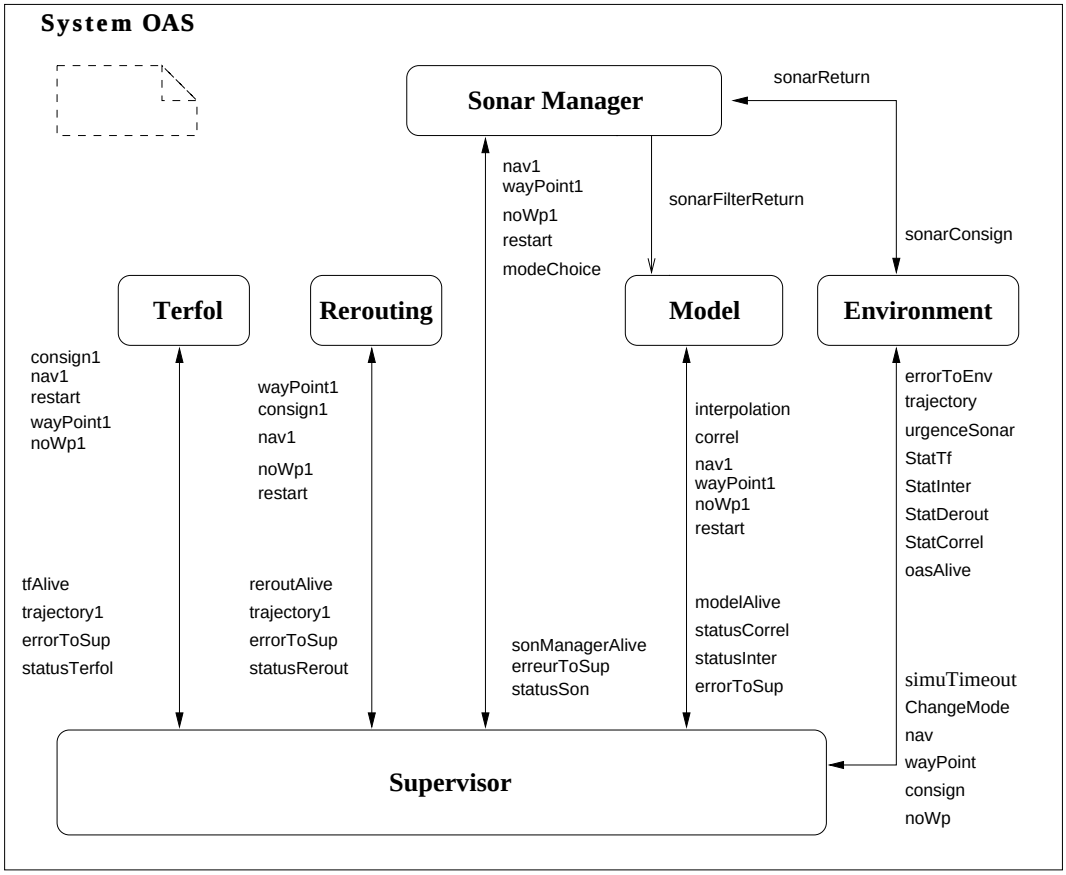

Fig. 7. SDL structure of OAS system

A.2 SDL environment process

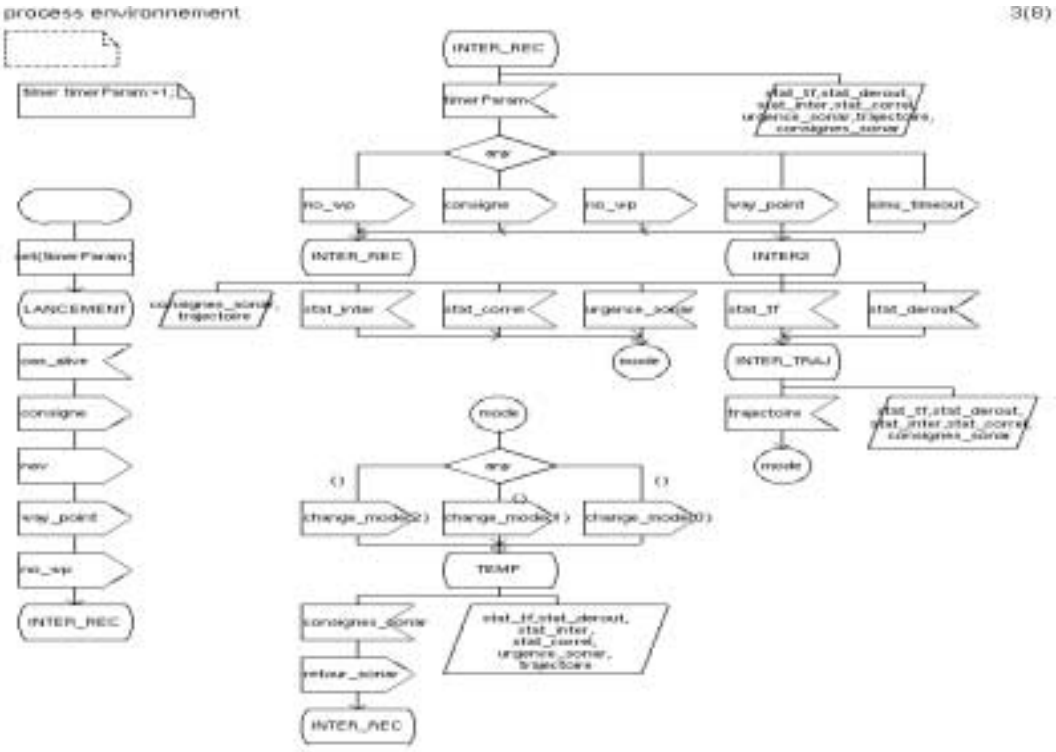

Fig. 8. SDL environment process (partially)