Model Checking SDL with Spin

Dragan Bošnački[1], Dennis Dams[2], Leszek Holenderski[1], and Natalia Sidorova[2]

[1] Dept. of Computing Sci., Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands
{dragan, lhol}@win.tue.nl

[2] Dept. of Electrical Eng., Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands
{dennis, natalia}@ics.ele.tue.nl

Abstract. We present an attempt to use the model checker Spin as a verification engine for SDL, with special emphasis put on the verification of timing properties of SDL models. We have extended Spin with a front-end that allows to translate SDL to Promela (the input language of Spin), and a back-end that allows to analyse timing properties. Compared with the previous attempts, our approach allows to verify not only qualitative but also quantitative aspects of SDL timers, and our translation of SDL to Promela handles the SDL timers in a correct way. We applied the toolset to the verification of a substantial part of a complex industrial protocol. This allowed to expose several non-trivial errors in the protocol's design.

1 Introduction

We present an approach to automating the formal verification of SDL, by model checking SDL specifications with Spin. SDL [8] is a visual specification language, especially well suited for communication protocols, and quite popular in industry. Spin [5] is one of the most successful enumerative model checkers.

In order to connect the Spin verification engine to SDL, we had to extend Spin in two ways. First, we had to implement a front-end which would allow to automatically translate SDL to Promela (the input language of Spin). Second, we had to extend Spin with the notion of discrete time, to be able to model SDL timers. The extended version is called DT Spin and its input language is called DT Promela (where DT stands for *discrete time*).

The translation of SDL to Promela is split into two steps. In the first step we use the `sdl2if` tool, implemented in Verimag, Grenoble, which transforms SDL programs to the intermediate format IF [3] that was designed for the representation of timed asynchronous systems. This first step flattens the hierarchic structure of SDL blocks to bare processes which can then be directly transformed to Promela processes, in the second step, by our tool `if2pml`.

* This research has been supported by the VIRES project (Verifying Industrial Reactive Systems, Esprit Long Term Research Project #23498).

We applied our method to the verification of a substantial part of MASCARA—a complex telecommunication protocol developed by the WAND (Wireless ATM Network Demonstrator) consortium [13]. As a result, we exposed several non-trivial errors in the design of MASCARA.

In order to resolve the usual problems caused by the lack of the formal semantics of SDL, we decided to rely on the semantics of SDL as determined by the ObjectGEODE tool [11]. In particular, we assume that transitions are atomic and instantaneous, and timeout signals are not necessarily sent at the beginning of a time slice (in other words, the timer messages are treated like other messages, without any special priority). More details in section 3.2.

We are aware of two other attempts to use Spin as a verification engine for SDL [6, 10]. In our opinion, they were not fully successful. First, both approaches tackle the qualitative aspects of SDL timers only, in the sense that they abstract out the concrete values of timers. Our approach allows to analyze the quantitative aspects of SDL timers as well. Second, the previous approaches are incorrect, as far as the timing issues are concerned. More precisely, instead of just introducing more behaviours, which is unavoidable when the concrete values of timers are abstracted out, they simultaneously remove some of the behaviours that are allowed by SDL, which may lead to unsound results (so called "false positives"). Some concrete examples are given in section 3.3. The incorrectness of the previous attempts also shows that taking the timing issues of SDL into account, when using Spin to model check SDL, is not trivial.

We do not claim that our approach is correct, in the formal sense. Ideally, one should prove that the approach is sound (no "false positives" are possible) and complete (no "false negatives" are possible). In principle, such a correctness result cannot be established, due to the lack of formal semantics, both for SDL and Promela, which would be simple enough to carry such correctness proofs. However, we give some informal justification of the correctness of our approach.

We clearly separate the qualitative and quantitative aspects of SDL timers. This allows to analyze the SDL models that use timers, both in the abstract and concrete way. The two methods have their own benefits and drawbacks. In the abstract case, if DT Spin decides that some safety property holds then the property is true for all values of timers, and is thus time independent. This may be a desired feature of a model. On the other hand, proving the time independence may come at a price: "false negatives" are possible, in the case a property does depend on time. The analysis with the concrete values of timers does not lead to "false negatives", but the price may be a bigger state space that must be enumerated by DT Spin.

We put some effort in making DT Spin a "conservative" extension of Spin: DT Promela is designed in such a way that standard Spin can be used to model check DT Promela programs obtained from the SDL models with abstracted timers. This may be useful for those who prefer to use a proven technology, instead of our experimental DT Spin.

The paper is organized as follows. In section 2, we give an overview of Spin and DT Spin. Section 3 is devoted to the translation of SDL to DT Promela. The verification method and its application to the MASCARA protocol is presented in sections 4 and 5. Finally, we conclude with section 6.

2 Spin and DT Spin

2.1 Spin and Promela

Spin [5] is a software tool that supports the analysis and verification of concurrent systems. The system descriptions are modelled in a high-level language, called Promela. Its syntax is derived from C, and extended with Dijkstra's guarded commands and communication primitives from Hoare's CSP.

In Promela, system components are specified as *processes* that can interact either by message passing, via *channels*, or memory sharing, via *global variables*. The message passing can either be buffered or unbuffered (as in Hoare's CSP). Concurrency is asynchronous (no assumptions are made on the relative speed of process executions) and modelled by interleaving (in every step only one *enabled* action is performed).

Given a Promela model as input, Spin generates a C program that performs a verification of the system by enumerating its state space, using a depth-first search algorithm. This way, both *safety* properties (such as absence of deadlock, unspecified message receptions, invalid end states, and assertions) and *liveness* properties (such as non-progress cycles and eventual reception of messages) can be checked. The most general way of expressing properties in Spin is via so-called *never claims*, which are best seen as monitoring processes that run in lock step with the rest of the system. The never claims are, in fact, Büchi Automata, and thus can express arbitrary omega-regular properties. Spin provides an automatic translator from formulae in linear-time temporal logic (LTL) to never claims, so it can be used as a full LTL model checker. In case the system violates a property, the trace of actions leading to an invalid state, or a cycle, is reported. The erroneous trace can be replayed, on the Promela source, by a guided simulation.

To cope with the problem of state space explosion, Spin employs several techniques, such as partial-order reduction, state-vector compression, and bit-state hashing.

2.2 DT Spin and DT Promela

DT Spin [2] is an extension of Spin with discrete time. In the time model used in DT Spin, time is divided into slices indexed by natural numbers that can be seen as readings of a fictitious global digital clock that ticks at the end of each slice. The events happening in the same slice are assigned the same clock value, so the elapsed time between events is measured in ticks. Since concurrency is modelled by interleaving, all the events happening in one run of a system are totally ordered and thus two events happening in the same slice are not considered necessarily simultaneous. Instead, they are considered to be ordered, and their ordering inside one slice is determined by the ordering of the run. The properties that depend only on the ordering of events are called *qualitative* while those depending on the elapsed time between events are called *quantitative*. In our model, time passes only if no other action in the system is possible.

In order to capture timing features, standard Promela is extended to DT Promela. A new data type, called `timer`, is introduced. It is used to declare variables that represent discrete-time countdown timers. Three new statements that operate on timers are added: `set(`*tmr*`,`*val*`)` activates the timer *tmr*, by assigning the integer value *val* to *tmr*, `reset(`*tmr*`)` deactivates *tmr*, by setting it to -1, and `expire(`*tmr*`)` tests whether the value of *tmr* is 0. Initially, a timer has value -1.

In fact, the new statements are defined as Promela macros, in a special header file included at the beginning of every DT Promela model:

```
#define timer           short /* a short integer */
#define set(tmr,val)    tmr = val
#define reset(tmr)      tmr = -1
#define expire(tmr)     tmr == 0
```

The new statements allow to model a broad class of timing constraints, and other timed statements can easily be defined as Promela macros, by combining `set`, `reset` and `expire` with the control flow statements offered by Promela. There is yet another operation on timers: the `tick` statement decreases the value of all active timers by 1. It is used internally by DT Spin, at the end of every time slice, and is not available to the user.

DT Spin is fully compatible with Spin, and all features of Spin can be used to analyse discrete-time models. In particular, the partial order reduction algorithm of Spin [7, 9] had to be adapted for timed systems [2]. Besides qualitative properties, a broad range of quantitative properties can be verified using boolean expressions on timer values, in the assertions and LTL formulae.

3 Translating SDL to DT Promela

The process of model checking an SDL specification is depicted in Figure 1. An SDL specification is pushed through the pipe of translators `sdl2if` and `if2pml`, to obtain a DT Promela program that serves as input to `dtspin` or `spin`. The result of a negative verification experiment (e.g., an erroneous trace) has to be checked manually against the SDL specification.

`sdl2if` translates SDL to the language IF (Intermediate Format, [3]) which is a specification language for timed concurrent systems consisting of a fixed number of communicating automata. IF was designed as an intermediate formalism for connecting several industrial formal description techniques, such as LOTOS and SDL, to a number of verification tools developed in the research community.

`sdl2if` is implemented with the help of the SDL/API Interface provided by the ObjectGEODE tool [11]. The current implementation of `sdl2if` is able

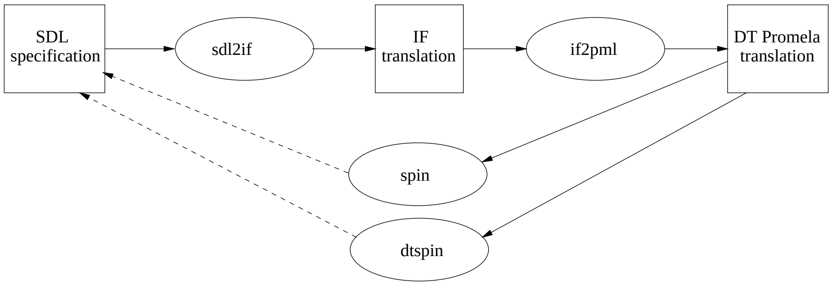

Fig. 1. The tools used in the verification.

to translate a substantial subset of SDL. One of the omissions are dynamically created processes.

In the overall SDL to DT Promela translation, `sdl2if` resolves the hierarchical aspects of SDL by "flattening" the hierarchy of blocks down to bare processes and resolving appropriately the sources, destinations and priorities of the signals exchanged between the processes. On the way, the SDL procedures are eliminated by inlining (when possible), and the syntactic sugar of SDL (enabling conditions, continuous signals, the special input *none*) is transformed to more primitive constructs. Moreover, some implicit constructs are made explicit (sender, offspring, parent, discarded signals). Details are given in [3].

On the other hand, `if2pml` performs the translation of the SDL core language, coping with issues that have more semantical flavor (like preserving the atomicity of the transitions or implementing the discard mechanism). Since IF is intended to be an intermediate language for a variety of high-level specification formalisms, it is quite expressive. As not all of its constructs are needed for representing SDL models, `if2pml` only translates the subset of IF that is relevant to SDL.

IF and `sdl2if` were developed at Verimag, Grenoble, while `if2pml` and `dtspin` were developed at the Eindhoven University of Technology. All the tools were developed in the framework of the VIRES project [12].

In what follows we describe the translation from IF to DT Promela in more detail. The presentation is divided into three parts. In section 3.1 we describe how the SDL processes (i.e., IF automata) are represented in Promela. In section 3.2, the DT Promela representation of the SDL/IF timers is given. Finally, in section 3.3, the abstraction from the concrete values of timers is described.

3.1 IF to Promela: translating automata

As in Promela, the IF models are sets of processes that communicate via buffers. This provides an almost one-to-one translation of these concepts. Also, the IF data types can directly be translated to their Promela counterparts (with some minor restrictions on the range types).

The way the SDL/IF automata are represented in Promela is fairly standard, and can be grasped by comparing the IF source given in Fig. 2 with its Promela translation in Fig. 3. A state is represented by a Promela label. All the outgoing transitions are translated to the branches of the choice statement associated with the label. The discard mechanism is implemented via self-looping to the same state, after reading a signal that has to be discarded in the state. The atomicity of SDL/IF transitions is preserved by putting the `if` statement inside the `atomic` statement.[1]

```
process proc: buffer buf;
state
   state1 discard sig3, sig4 in buf
   state2
   ...
transition
   from state1 input sig1 from buf do body1 to state2;
   from state1 input sig2 from buf do body2 to state3;
   ...
```

Fig. 2. A skeleton of an IF program.

```
proctype proc() {
  state1: atomic {
    if
    :: buf?sig1 -> translated_body1; goto state2;
    :: buf?sig2 -> translated_body2; goto state3;
    ...
    :: buf?sig3 -> goto state1; /* discard */
    :: buf?sig4 -> goto state1; /* discard */
    fi
  }

  state2: atomic{
    ...
  }
  ...
}
```

Fig. 3. Promela translation of the structure from Fig. 2

The implementation of `if2pml` is still in a prototype stage and some SDL features are not supported yet. The most notable omissions are the mechanism of

[1] A special care is taken to correctly handle the stable and non-stable states in IF.

saving a signal, the dynamic creation of processes, and the abstract data types. The implementation of the save mechanism in Promela is possible, but rather involved. It may lead to a substantial growth of the state space during model checking, and for this reason we have chosen to omit it. The dynamic process creation is a basic feature of Promela, so it can easily be implemented once `sdl2if` supports it. Finding a satisfactory solution for the abstract data types remains a future work for both `sdl2if` and `if2pml`.

3.2 IF to DT Promela: translating timers

The crucial issue about time in SDL is the detection of timer expirations (timeouts). In SDL, a timer expiration results in sending a timeout pseudo-signal to the input queue of the process the timer belongs to. The timeout signal is then handled as an ordinary signal: it is either consumed, by a transition that is guarded by the timeout signal, or discarded, in a state with no such transitions. In both cases, the corresponding timer is deactivated.

Our implementation of timers does not use such timeout signals. Instead, we associate with each SDL timer a DT Promela timer variable, and detect the expiration of the timer by testing whether the timer variable has value 0, and the timer is deactivated by setting the timer variable's value to -1.

More precisely, for each timer *tmr* declared in an SDL process, we add a new branch to all the choice statements associated with states (see figure 3). Assume a state, say `state1`, with an outgoing transition guarded by *tmr*. For such a state we add the branch

```
:: expire(tmr) -> reset(tmr); translated_body1; goto state2;
```

If `state1` has no outgoing transitions guarded by *tmr*, we add the branch

```
:: expire(tmr) -> reset(tmr); goto state1; /* discard */
```

It turns out that under the time semantics given below (as determined by the ObjectGEODE tool [11]), these two approaches to timers are equivalent. However, the "variable" approach has an advantage over the "signal" approach, from the verification point of view, since it generates smaller state spaces. In the "signal" approach, an additional Promela process (or even several processes) would be needed, in order to generate the timeout signals in a right way. This, together with the overhead of exchanging timeout signals, increases the state space.

In what follows, we give an informal justification of the above mentioned equivalence.

The semantics of time in SDL. Transitions are instantaneous. Time can only progress if at least one timer is active and all SDL processes are waiting for further input signals (i.e., all input queues are empty, except for saved signals). Time progression amounts to performing a special transition that makes time increment until an active timer expires. In the sequel, we refer to the segments of time separated by the special transition as *time slices*. (Note that time progression is discretized.)

With each timer there is associated a pseudo-signal and an implicit transition, called a timeout transition. When a timer expires, in some time slice, its timeout transition becomes enabled and can be executed at any point of the time slice. The execution of this transition adds the associated pseudo-signal to the process queue. The timeout transitions of the timers that expire simultaneously can be executed in any order.

If the `set` or `reset` operation is performed on a timer after its timeout transition becomes enabled, the timeout transition is disabled. If the timer is set or reset after adding its associated pseudo-signal to the process queue, the pseudo-signal is removed from the queue.

Model equivalence. In order to justify the equivalence of the two models we need to show that any execution sequence in the signal model can be simulated by an execution sequence in the variable model, and vice versa. In what follows, we assume that SDL timers are never set to the special value *now* (i.e., our timer variables are never set to 0, explicitly), and we only concentrate on the simulation of the transitions which relate to timers, since the two models coincide on the untimed features. There are two issues which have to be considered: the `set` and `reset` operations on timers, and the expiration of timers.

The `set` and `reset` operations coincide in the two models, so this issue does not cause problems. As far as the expiration of timers is concerned, it should be obvious that the time slice in which a timer expires is recognized in the same way, in both models.

The only problematic issue is whether consuming/discarding the timeout signals, in the signal model, is properly simulated with our count-down timers, and vice versa. Our claim that, in fact, this is the case is based on the assumption that the following, more direct, translation of SDL/IF to Promela would be correct.

Assume ts_T denotes the timeout signal corresponding to timer T, in the signal model. In order to handle the consumption of ts_Tlike the consumption of an ordinary signal, by an SDL/IF transition guarded by ts_T, the following branch

```
:: buf?ts_T -> translated_body1; goto state2;
```

could be added to the Promela translation in figure 3.

Similarly, in order to handle the discarding of ts_T as an ordinary signal, the following branch

```
:: buf?ts_T -> goto state1; /* discard */
```

could be added to the choice statements that corresponds to the states with no outgoing transitions guarded by ts_T.

Observe that the

```
:: expire(tmr) -> reset(tmr); ...
```

branches, in our real Promela code, correspond directly to the above branches. More precisely, `expire(tmr) -> reset(tmr)` corresponds directly to `buf?ts_T`. Namely, `expire(tmr)` corresponds to the check whether *ts_T* is in the input queue, and `reset(tmr)` corresponds to the removal of *ts_T* from the queue.

3.3 Timer abstraction

It turns out that standard `spin` can also be used to model check a DT Promela model, for a property that does not depend on time. The advantage of using `spin` instead of `dtspin` is that `spin` usually consumes less resources than `dtspin`. In order to convert a DT Promela model into a Promela model, it suffices to change the special header file included at the beginning of every DT Promela model (see section 2.2):

```
#define timer          bool
#define set(tmr,val)   tmr = true
#define reset(tmr)     tmr = false
#define expire(tmr)    tmr == true
```

The new definitions abstract out the concrete values of *active* timers, by consistently mapping them to `true`. The concrete value -1, which is used to mark an *inactive* timer, is consistently mapped to `false` (under the assumption that each timer is initialised to `false`). Obviously, such an abstraction is sound since no behaviours are lost. More precisely, any behaviour with concrete timers can be simulated with abstract timers, since the active timers are allowed to expire nondeterministically, at any time.

This is not the case in the related approaches [6, 10], where some heuristic approximations of the timer behaviour are used rather than a proper abstraction. The approximations do not only add some behaviours, but also loose some of them, as shown in the following examples.

In [6] the authors try to minimize the number of timer expirations, in the abstract system, that do not correspond to expirations in the concrete system. They take advantage of the observation that, in practice, the transitions guarded by a timer expiration are supposed to resolve the kind of deadlocks when no other transitions except the ones triggered by timeout signals would be enabled. The timers are represented by processes that send the corresponding timeout signals when the special Promela statement `timeout` becomes enabled. The `timeout` statement becomes enabled when no other transition is enabled, in the system.

However, there are situations when a behaviour of the concrete system cannot be simulated by the approximate one. Let us consider two concurrent processes, say P_1 and P_2, specified by the following transition systems:

$$P_1 : \bullet \xrightarrow{T_1/A} \bullet \xrightarrow{B/} \bullet$$

$$P_2 : \bullet \xrightarrow{T_2/B} \bullet \xrightarrow{A/} \bullet$$

where T_1 and T_2 are timer signals, A and B are normal signals, and X/Y denotes "receive X and send Y to the other process".

If both T_1 and T_2 expire simultaneously then, in the concrete system, P_1 and P_2 may exchange signals A and B, and both reach their final states. However, in the abstract system, one of the processes will not be able to reach its final state. Initially, the first transitions of both processes are enabled. If P_1 performs its first transition (and thus sends A to P_2), the first transition of P_2 becomes disabled. P_2 must discard A, before its first transition becomes enabled again, and thus it will not be able to pass the $A/$ guard. Similarly, if P_2 performs its first transition, P_1 will not be able to pass the $B/$ guard.

In the approximation used in [10], one pseudo-process is used to handle all timer expirations. After each `set` operation this pseudo-process immediately sends the corresponding timeout signal to the input queue of the process that sets the timer. As a result, if an SDL process uses two timers, they can only expire in the order they are set, no matter what values they are set to. Obviously, the behaviour in which an SDL process sets the timers T_1 and T_2 (in that order) to such values that T_2 expires before T_1 cannot be simulated, by the abstract system.

4 Verification methodology

The aim of this section is to present the way the real verification process is performed. Industrial-size SDL models normally cannot be verified with existing model-checking tools as a whole, so the first natural task is to split a protocol in some relatively autonomous parts of a reasonable size and apply to them compositional verification. Fortunately, due to their block-structure, SDL-systems can be usually split in a natural way without much effort.

The obtained sub-models are not self-contained, i.e. the behaviour of their environment is not specified. Since `spin` can only model-check closed systems we need to close our model first. We achieve this by adding an "environment" process specified in SDL at the same hierarchy level as the extracted model itself.

The simplest possible approach is to construct an environment producing all the "possible" behaviours but practice shows that this is not of much use in real life. Such an environment leads to adding to the model too many erroneous behaviours and thus one gets too many "false negatives" during model-checking. Local livelocks, cycles with non-progressing time, and non-existing deadlocks are the typical examples of those false-errors. Moreover, since many redundant behaviours are added, this may also lead to a state explosion.

Another possibility is to construct an environment being able to send/receive a signal whenever the modelled system is ready to get/send it. Applying such an approach reduces the added behaviours but it still adds some unwanted behaviours caused by sending non-realistic signal sequences.

Both these approaches are not safe: in case of adding non-progressing time cycles, we loose some behaviour of the system. So they can be considered as a kind of heuristics only, that may be of some use at the first stage of system debugging.

A different approach is to provide an SDL-specification of the "right" environment, i.e. the one, which faithfully models the assumptions under which the component was designed, giving an abstraction of a real environment. Although it makes the soundness of verification results dependent on the quality of the environment model, it usually turns out to be a practical method.

The closed SDL-model can be automatically translated into DT Promela through the translators sdl2if and if2pml. Then one should choose between verification of the concrete model with `dtspin` and verification of the model with abstracted time in the standard `spin`. First, the built-in `spin` features (finding deadlocks and livelocks, unreachable code) are used. After correcting discovered structural errors, functional properties defined by a designer or drawn from the informal specification of the system can be verified.

It would seem obvious to verify all non-timed properties with an abstracted-time model and all timed properties with a concrete model. However, sometimes it is more convenient to verify non-timed properties with a concrete model as well. If some functional property was proved with the abstracted-time model, it is proved for all possible values of timers. However if it was disproved, or a deadlock in the model was found, the next step is to check whether the erroneous trace given by `spin` is a real error in the system or it is a false error caused by adding erroneous behaviour either with abstracting from time or with too abstract specification of the environment. It can happen that the property does not hold for the concrete model, however the erroneous trace given by `spin` is one of the added behaviours. This behaviour cannot be reproduced for the SDL model with SDL-simulation tools and we cannot conclude whether the property holds or not.

One can not force `spin` to give the trace from the non-added behaviours. `dtspin` allows to reduce the set of added behaviours guaranteeing that timers are expiring in the correct order. In our verification experiments we had a number of cases when application of `dtspin` instead of `spin` gave a chance to get a real erroneous trace and disprove the property (see the next section).

5 Case study: verifying the MASCARA protocol

We have applied our tools to the verification of an industrial-size communication protocol called MASCARA (Mobile Access Scheme based on Contention and Reservation for ATM), developed by the WAND (Wireless ATM Network Demonstrator) consortium [13]. The protocol is an extension of the ATM (Asynchronous Transfer Mode) networking protocol to wireless networks.

A wireless ATM network comprises of a fixed wire-based ATM network that links a number of geographically distributed access points, which transparently extend ATM services to mobile users, via radio links operating in the 5.2 GHz frequency band and achieving data rates of 23.5 MBits/s. The task of MASCARA is to mediate between the mobile users and the access points, to offer the mobile users the advantages of the standard wire-based ATM protocol: high reliability, capacity-on-demand, and service-independent transport. The transparent extension of ATM over radio links is a challenge, as standard ATM has not been designed to work in a wireless environment. The radio medium is characterised by a high bit error rate, the transmission mode is typically broadcast (or at least multicast) and the scarcity of available radio bandwidth calls for a time division duplex (i.e. half duplex) mode. All this leads to the necessity of extra functionality allowing to cope with these dissimilar environmental properties.

A crucial feature of MASCARA is the support of *mobility*. A mobile terminal (MT) can reliably communicate with an access point (AP) only within some area called the AP's *cell*. Whenever an MT moves outside the cell of its current AP it has to perform a so-called *handover* to an AP whose cell the MT has moved into. A handover must be managed transparently with respect to the ATM layer, maintaining the agreed quality of service for the current connections. So the protocol has to detect the need for a handover, select a candidate AP to switch to and redirect the traffic with minimal interruption.

The protocol is too large to be automatically verified as a whole so we have concentrated our efforts on the verification of a substantial part called MCL (MASCARA Control). The main purpose of MCL is to support the mobility issues. It contains 9 processes, each with up to 10 states (6 on average). Its main function is to monitor the current radio link quality, gather information about radio link qualities of its neighbouring APs (to make a quick handover, in the case of deterioration of the current link quality), and switch from one AP to another (during the handover procedure).

During the first phase of the verification, several deadlocks were found. Most of them were related to improper synchronisation between various request/confirm subprotocols (a component requests a service from another component and waits until the confirmation arrives that the request is either rejected or served). The second source of deadlocks was the potential race conditions between various components of MCL, due to the fully asynchronous communication in SDL (non-blocking sending of messages). The following example describes one of the race conditions we have found.

Most MASCARA components must be initialised before they are prepared to serve requests from other components. The components are grouped in various tree-like hierarchies, and the initialisation phase for a group is triggered by sending the INIT signal to the group's root. Each node that receives the INIT signal resends the signal to all its children. However, in such a cascade of INIT signals there is a possible race condition: `spin` found a trace in MCL where an already initialised component tried to immediately request a service from a component that had not been initialised yet. Such a request was simply discarded and thus its confirmation was never sent back, leading to a deadlock in MCL.

After correcting these errors, we continued the verification, by exploiting another useful `spin` feature—unreachable code diagnosis. The analysis of the unreachable code reported by `spin` revealed that some code for serving one particular request is never reached and thus the request for that particular service was never confirmed. Further analysis showed that there was a local deadlock possible. (This local deadlock could not be discovered by `spin`, in previous experiments, since `spin` can only discover global deadlocks.)

Finally, we verified some functional properties that we derived from the informal specification. An example of such a property is given below.

One of the tasks of MCL is to periodically update a list that contains information about the quality of radio links with the neighbouring APs. This updating phase is called the *TI procedure*. In order to judge the quality of a radio link with a particular AP, MCL must connect to that particular AP, so the connection with the current AP is suspended during the whole TI procedure. Therefore, another procedure, checking the quality of the current connection and making the decision on the necessity of a handover, should not interfere with TI procedure.

This requirement was encoded as a simple observer for the following safety property: a handover request never comes in between the two messages that mark the beginning and the end of the TI procedure. The verification experiment with `spin` which was supposed to confirm this property, showed instead a trace violating it. Unfortunately, the erroneous trace could not be simulated by the SDL model of MCL (so we got a so-called "false negative"). However, when exactly the same experiment was repeated with `dtspin` we got a different erroneous trace which could then be simulated by the SDL model. Thus, the property was indeed violated, `dtspin` allowed us to find yet another error in MASCARA protocol.

The explanation of the different results obtained with `spin` and `dtspin` is obvious. The TI procedure is triggered by a timer, so the behaviour of the TI protocol could indeed depend on proper timing. In the model with abstracted time, as used in `spin`, timers can expire at any time, so `spin` can produce a wrongly timed trace that is not accepted by an SDL simulator (which allows only properly timed traces, of course). After finding a way to re-design the components, some other functional properties were proved.

When we planned the first series of verification experiments we expected to reach the limits of `spin` and `dtspin` quickly. We were proved wrong. In the experiment that consumed the most resources, `spin` reported the following statistics:

```
State-vector 416 byte, depth reached 3450, errors: 0
  55959  states, stored
  23.727 memory usage for states (in MB)
  25.582 total memory usage (in MB)
  14.2   total time (in seconds)
```

which should be read in the following way:

`State-vector 416 byte` is the memory needed to represent one state.

`55959 states, stored` is the number of different states found in the model (all the states must be kept during the state space enumeration, so 23.727MB memory was needed for states).

`depth reached 3450` is the greatest depth reached during the depth-first search (since `spin` must keep a state stack of at least this depth, about 1.7MB was needed in addition to the state memory).

It is quite likely that with our 2048MB memory we will be able to handle even more complex case studies.

6 Conclusions and future work

We have developed a tool, `if2pml`, that enables the translation from SDL to Promela. It can be used, together with the accompanying tool `sdl2if`, to model check SDL specifications with Spin.

Our approach preserves the timing aspects of SDL. This is in contrast to other translators that we know, which only approximate timing aspects. SDL timers, which expire by sending signals, are in our approach translated into the timer variables provided by DT Promela. We have argued the correctness of this translation.

The tool has been successfully used on an industrial case study. It is publicly available from `http://radon.ics.ele.tue.nl/~vires/public`.

As a future work, we consider to extend the tool by implementing the translation of dynamic process creation in SDL and the `save` construct. SDL supports various styles of specifying data types. It needs to be investigated how the specification of data aspects can be combined with the translation from SDL to Promela.

Acknowledgments

We gratefully acknowledge VERILOG for giving us free access to their *ObjectGEODE* tool. We would also like to thank Marius Bozga, Lucian Ghirvu, Susanne Graf (Verimag, Grenoble) and Gerard Holzmann (Bell Labs) for fruitful discussions during the implementation of the `if2pml` tool.

References

1. R. Alur, D.L. Dill, *A Theory of Timed Automata*, Theoretical Computer Science, 126, pp.183-235, 1994.
2. D. Bošnački, D. Dams, *Integrating Real Time into Spin: A Prototype Implementation*, S. Budkowski, A. Cavalli, E. Najm, editors, Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'98), Kluwer, 1998.
3. M. Bozga, J-C. Fernandez, L. Ghirvu, S. Graf, J.P. Karimm, L. Mounier, J. Sifakis, *If: An Intermediate Representation for SDL and its Applications*, In Proc. of SDL-FORUM'99, Montreal, Canada, 1999.
4. I. Dravapoulos, N. Pronios, S. Denazis *et al.*, *The Magic WAND, Deliverable 3D2, Wireless ATM MAC*, September 1997.
5. G. J. Holzmann, *Design and Validation of Communication Protocols*, Prentice Hall, 1991. Also: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`
6. G.J. Holzmann, J. Patti, *Validating SDL Specification: an Experiment*, In E. Brinksma, G. Scollo, Ch.A. Vissers, editors, Protocol Specification, Testing and Verification, Enchede, The Netherlands, 6-9 June 1989, pp. 317-326, Amsterdam, 1990. North-Holland.

7. G.J. Holzmann, D. Peled, *An Improvement of Formal Verification*, PSTV 1994 Conference, Bern, Switzerland, 1994.
8. A. Olsen *et al.*, *System Engineering Using SDL-92*, Elsevier Science, North-Holland, 1997.
9. D. Peled, *Combining Partial Order Reductions with On-the-Fly Model Checking*, Computer Aided Verification CAV 94, LCNS 818, pp. 377-390, 1994.
10. H. Tuominen, Embedding a Dialect of SDL in PROMELA, 6th Int. SPIN Workshop, LNCS 1680, pp. 245-260, 1999.
11. Verilog, *ObjectGEODE SDL Simulator - Reference Manual*, `http://www.verilogusas.com/solution/pages/geode.htm`, 1996.
12. VIRES, *Verifying Industrially Relevant Systems*, Esprit Long Term Research Project #23498, `http://www.win.tue.nl/cs/fm/vires`, 1996.
13. WAND consortium, *Magic WAND - Wireless ATM Network Demonstrator*, `http://www.tik.ee.ethz.ch/ wand`, 1996.