Formal Verification of SDL Systems at the Siemens Mobile Phone Department

Franz Regensburger * and Aenne Barnard **

Siemens AG

Abstract. In this paper we present the multi process verification of SDL systems at the Siemens mobile phone department. The formal verification is efficiently performed by the model checker tool SVE that is developed at Siemens, too. Several case studies in the area of telecommunication protocols show that model checking techniques have reached a state where they can profitably be integrated in the industrial software development process.

Keywords: formal verification, model checking, SDL, telecommunication protocols.

1 Introduction

Specification of telecommunication software with SDL-like languages has a pretty long tradition at Siemens. Besides the standardised SDL versions SDL-like languages have been used at least for the last 20 years at the departments that develop software for telecommunication.

In the last years formal verification through model checking has gained increasing attention in industries since this kind of formal verification has proved its effectiveness not only in academic case studies.

A research and development group at Siemens ZT AN 1 is working on automatic verification of SDL processes since 1993. In the beginning verification concentrated on the model checking of single SDL processes. However, during the past two years the model checker group at Siemens ZT AN 1 has worked on several case studies in cooperation with the Siemens mobile phone department to extend the application of model checking to multiprocess systems.

This paper reports on the results of joint work between Siemens PN KE as the user of formal verification and Siemens ZT as tool supplier. The case studies show that model checking effectively integrates in the software development process and demonstrably improves software quality.

The paper is organised as follows. After some preliminary remarks in section 2 about the GSM case study and the operating system of a mobile phone designed at Siemens we briefly survey the technical background of the verification tool SVE in section 3. Section 4 explains why the specification language SDL is used at Siemens.

* Franz.Regensburger@mchp.siemens.de

** Aenne.Barnard@pn.siemens.de

Section 5 is devoted to the task of automatic extraction of a verification model from an SDL description. Section 6 shows some examples for system properties formulated in the property language SPL and section 7 shows the usage of the tool SVE in more detail. In section 8 we report time and space complexity for two case studies performed at the Siemens mobile phone department. Section 9 concludes the paper with some remarks about current and future work.

2 Protocol Software for Mobile Phones

For a couple of years mobile phones are getting more and more popular around the world. Besides basic telephone functionality additional features like short messages, faxes or data transfer are standard by now. Although so small and handy every mobile phone contains a lot of software (about 500 000 loc) which is an implementation of the highly complex GSM protocol.

The GSM protocol is organised in an architecture structured into three layers. In the application of formal verification we have mainly concentrated on layer 3. The GSM layer 3 is responsible for linking the mobile to the network if possible. For that it has to trace the surrounding base stations continuously, it has to determine which one of these bases is the best and finally make sure that the mobile is connected to that specific base. In parallel incoming and outgoing calls must be controlled. An implementation of layer 3 has to make sure that these functions synchronise properly without running into life- or deadlocks.

The connection of the processes we concentrated on in our case studies is shown in figure 1. The examples in section 8 correspond to different versions of the depicted scenario. An important process of layer 3 is the mobility manager MM. One of its slave processes is the radio resource manager RR, which is responsible for establishing and maintaining connections. During idle mode (no call is active) RR controls the two slave processes cell selection CS and cell reselection CR.

The process CS establishes the connection to some base station (cell). To do so it first looks for radio waves in the air and then filters out the waves that really correspond to base stations. Among these stations it selects the one, which is the best according to some metrics that considers not only the quality of the signal but also roaming contracts between the net providers. When CS has established the connection it remains in some busy state. At that moment its master RR starts the process CR, which has to maintain the connection established by CS. The processes RR, CS and CR are able to send messages to the lower protocol levels 2 and 1. However, all answers from the lower levels are sent to RR, which routes packages addressed to CS and CR. See section 5.2 for the modelling of the processes and their environment.

The mobile software at Siemens is organised in a multi process single processor architecture, which means all the functions the software performs are split between independent processes but all the processes share a single processor. The processes can communicate with each other by messages. An incoming mes-

sage is stored temporarily in a queue until the process can respond. Due to our architecture at most one of the processes can use the processor at any time.

Fig. 1. Processes of GSM protocol layer 3 in idle mode

A process using the processor is called *active*. All the other processes are *inactive*. It is assumed that an active process becomes inactive again after some time. The inactive processes either have messages in their input queue, these processes are called *waiting*, or they do not have any message, these processes are called *idle*. The operating system performs a simple scheduling algorithm that determines which of the waiting processes becomes active next. See section 5.4 for the scheduling algorithm.

At the Siemens mobile phone department, during the design phase each process is understood as a simple state machine that has a number (normally 5-20) of states and sometimes a few very simple local data structures. In each state a process can react to input messages by modifying the values of local data structures and sending messages to other processes. Although this representation of the protocol is still quite abstract, it contains already the basic ideas, how the protocol should work and especially how the interprocess communication works.

3 The SVE System and Its Technical Basis

SVE (System Verification Environment) is a tool for the formal verification of software and hardware systems. It was developed at Siemens ZT AN 1. In SVE system properties are formulated in an application specific language SPL (SDL Property Language) [1] that is automatically translated into the temporal logics (A)CTL and LTL [3]. The system behaviour is modelled as a finite state machine that is automatically derived from an application specific system description, e.g. a SDL system description. The kernel of the SVE system is a BDD-based symbolic model checker, which was developed at ZT AN 1, too. Currently, there exist front ends for various SDL dialects, for control systems and for hardware description languages.

4 Using SDL as a Design Language

At the mobile phone department at Siemens we use SDL (Specification and Description Language) [2] as an input language. In the following we want to explain some of the reasons that lead us towards that decision.

The prime features we expected from our design language were:

- powerful enough to describe our design state machines
- widespread used in telecommunication
- tool support available
- easy to learn and easy to understand
- graphical representation available

At Siemens SDL is used in several departments as a design and as an implementation language. At the mobile phone department it is just used for the design at the moment. In the beginning our design models are very abstract and consist just of a few processes with some few states (see the examples in section 8). Therefore, a much simpler language would have done for the design. However, we plan to extend the use of SDL to the implementation phases as well, which requires a language with the expressiveness of SDL.

Software systems for telecommunication tend to be monsters in terms of size. The code grows over the years and changing the language base by a complete re-implementation is usually impossible. Therefore, the evolution of the languages used for design and implementation should be controlled by some institution like the ITU (International Telecommunication Union). Since SDL was designed especially for telecommunication and is controlled by the ITU it was quite natural to choose SDL.

Despite the standardisation by ITU SDL also has its weak points. The ITU publishes standards for the syntax and the semantics of SDL in working periods of currently 4 years. One goal of these standards is to guarantee portability between SDL tools. However, a recent tool evaluation at Siemens showed that several SDL tools, which officially support SDL/PR 92 or later, are only able to interchange SDL/PR 88 with one another. The situation on the semantic level is

even worse. Besides tool incompatibilities there is a widespread practice to use special purpose code generators instead of the generators supplied with the SDL tools.

At the mobile phone department we use the tool Object/Geode from Verilog with SDL/PR 92. Unfortunately the conformance with ITU only holds for the syntactic level. On the semantic level, our developers insisted in slightly changing the semantics of some SDL symbols and the scheduling policy. In section 5 we describe these changes and how we reflected them in the verification models generated by SVE (see especially sections 5.3 and 5.4).

5 Automatic Generation of the Verification Model

In this section we present the automatic generation of the verification model from the SDL input files. SVE has interfaces to various SDL dialects. For the multi process verification presented in this paper we use the phrase language SDL/PR of SDL 92 [2] that is standardised by the ITU (formerly CCITT).

5.1 Supported Sub-Language of SDL

At the mobile phone department at Siemens formal verification is used in the early design phase of a protocol. As a consequence, the SDL systems that need to be verified are on a rather abstract level. Besides message passing the processes just maintain some local variables that play the role of flags. Therefore, it is possible to restrict SDL to a large degree. In the following we enumerate the most important restrictions we imposed on the SDL/PR 92 input language.

Currently, the only data types that need to be supported are enumeration types with equality and inequality as atomic predicates. This is sufficient to model the flags mentioned above. However, the next release will also support finite integer ranges with appropriate arithmetics. Consequently the only tasks that are supported in transitions are assignments to local variables. All other tasks are simply ignored. At the time of this writing we are coding a version of the model generator that supports recursion free procedures and macros, too.

Of course selections and decisions are supported. Conditions that are only composed of atomic predicates about enumeration types are fully supported in their semantics. Selections and decisions that contain other conditions, e.g. informal text, are treated as nondeterministic branchings.

The provided and save directives are fully supported. However, our developers at the mobile phone department use a slightly modified semantics for these constructs.

Since our case studies did not use any of the object oriented features of SDL 92 we completely discarded the entire OO stuff.

Message passing operates on a restricted broadcast mechanism. Every process that mentions a specific signal in its input interface (in the type of some incoming signal route) will receive this signal if the sender is not explicitly mentioned with the send operation. In order to simplify the interpretation of the SDL input

we currently do not support the qualification of the receiver via signal route specification with the sending operation.

All the restrictions mentioned above were possible due to the abstract nature of the designs that are written by our developers. With the exception of recursion free procedures these restrictions were not a prerequisite for verification.

However, in order to enable the symbolic execution of transitions described in section 5.5 we had to inhibit procedures with recursion and the join construct. Another restriction vital to our approach was the inhibition of dynamic creation of processes.

5.2 Modelling the Environment

When we talk about multi process verification we actually mean the verification of some processes of a system that are communicating with each other. Due to the inherent complexity of this task we are usually not able to verify all the processes of an entire system at once.

However, we can verify smaller groups of processes which are closely coupled. Especially in the protocol area of telecommunication not all processes need to communicate with each other. Therefore, it is usually not difficult to identify such groups.

Of course, these groups are not isolated but communicate with other groups and perhaps with the environment of the entire system. It is a matter of taste whether the environmental groups and the systems environment are modelled as state machines, or whether their behaviour is modelled using temporal logic.

In practice, it turned out that all the complicated aspects of the environment should be specified via small pseudo SDL processes. Specifying the same behaviour via temporal logic is usually much more difficult and more error prone. Only if the environmental behaviour can be expressed easily by temporal logic, we use temporal formulae and add them as assumptions to our verification model. The fact that engineers have a very natural understanding of state machines but hesitate to use temporal logic was another encouragement to use this technique of environment specification.

Figure 2 shows this kind of modelling for one of our case studies. We concentrated on the processes RR, CS and CR of layer 3 of the GSM handy protocol (see section 2). The processes of layers 1 and 2 are modelled via a single pseudo SDL process LL. User interaction and relevant parts of the mobility manager were subsumed by the pseudo SDL process MM. Minor effects like timer interrupts and reset messages were modelled using temporal formulae that were added as assumptions.

5.3 Priority Queueing

In this subsection we concentrate on the queueing of messages. Standard SDL propagates *infinite* FIFO queues¹, which is obviously not adequate for small

¹ in this context infinite means: *there is always an available slot in the input queue.*

Fig. 2. Verification model for the GSM case study

hardware devices like mobile phones. Even worse, in the case study about mobile phones the system designers decided to use priority queues of a small finite size.

As a consequence our verification model has to perform some kind of overflow detection and should reflect priority queueing. Where the first obligation is not a problem the second one turned out to be hairy. Priority queues can either be coded using one FIFO queue per priority or by using a single sorted queue, where every buffer operation respects sorting. Both approaches tend to be rather space consuming in terms of BDD sizes. In the end we chose sorted queues as representation. For the variable ordering of the buffer bits the bitwise interleaving $b_{(1,1)}, \dots, b_{(n,1)}, \dots, b_{(1,m)}, \dots, b_{(n,m)}$ turned out to be best, where $b_{(k,l)}$ denotes bit l of buffer slot k .

5.4 Priority Scheduling

In section 2 we explained that it is assumed that an active process becomes inactive again after some time. This assumption of cooperative processes is technically accomplished in the following way. Whenever a process becomes active

it performs exactly one transition from its current SDL state to the next SDL state. When it reaches the next SDL state it becomes inactive again. During the transition it cannot be interrupted in any way no matter how long the execution of the transition may take.

A unique priority is assigned to every process of the system. Therefore, the scheduling algorithm is pretty easy: If more than one process has a message in its queue exactly that one with the highest priority becomes active.

This scheduling strategy together with the cooperative nature of processes influences the verification model to a large degree. One obvious consequence is that the transition of a process from one SDL state to the next one is an indivisible unit in every execution sequence of the system. This fact is heavily exploited as we will see in the next section.

5.5 Symbolic Execution

When coding the verification model one of the main questions is how fine grained the processes should be represented in the finite state machines. One first approach could be to represent every single task on an SDL transition as an execution point in the state machine. Since dealing with multiple processes means building the product automaton of all the automata for the single processes, this approach would yield gigantic BDD representations and therefore is not practicable.

Fortunately, the scheduling policy for the mobile phone operating system allows a far coarser granularity. Since a process cannot be interrupted during the execution of a transition, the verification properties for multi process verification usually do not depend on particular positions within a transition. They rather depend just on the current state, the current input message and the setting of local variables for the various processes.

Therefore, the automatic generation of the verification model performs a detailed analysis of the SDL description for every process. For every combination of state and input we compute every possible thread of execution that can be obtained from the SDL flow charts.

Next, we perform a symbolic execution of these threads. On every thread we accumulate changes in the local variable environment, output directives and logical conditions at branching points that are evaluated according to the actual symbolic variable environment. As a result we obtain the cumulated effect of every thread together with a set of logical conditions enabling the thread.

In the resulting state machine of a single process the abstract states² consist of the current SDL state, the local variable environment and the current buffer contents of the process. The abstract states of the state machine are connected through the computed threads above.

After we have computed the state machines for the single processes we finally combine them all with the simple scheduler described above by simply forming

² we use the term *abstract state* to distinguish the states in the state machine of the verification model from states in the SDL description.

the product automaton. In a last step we perform a complete reachability analysis of the entire system in order to speed up the model checking process, which takes advantage of the precomputed set of reachable states.

With the help of symbolic execution of threads and the ordering of BDD variables described above we are able to build verification models with reasonable space and time complexity³. From the above it should be clear that this is only possible due to radical optimisation techniques that are based on the special semantics of the operating system in the mobile phone case studies.

6 The Property Language SPL

In the last section we explained how the SDL input is translated into a verification model. In this section we briefly describe the language that is used to formulate properties about the system.

6.1 Technical Background

The property language SPL (SDL Property Language) [1] is translated into a language of temporal logic, which then in turn is fed to the model checker. Properties are translated into the temporal logic CTL [3] whereas assumptions are translated to LTL [3].

Since the generated verification model is an abstract view of the real system it usually allows more computation sequences than the real system. This is the reason why we restrict our property language to the universal fragment ACTL⁴ of the logic CTL [3, 4]. Strictly speaking, the constructs of our property language SPL are defined in a way that their translation to CTL always yields formulae of the fragment ACTL.

6.2 Some Properties Formulated in SPL

Now we are going to give some typical examples for properties and their SPL representations. Representative examples of system properties are:

1. Process p_1 and process p_2 are never in their busy states at the same time.
2. There is never an overflow in the input queues for any of the processes.
3. If process p_1 receives a `stop_req` this process eventually reaches its idle state

SPL properties look like simple english sentences. Very often a property formulated in natural language can be transformed into a SPL formula quite easily. The properties listed above are expressed in SPL as follows:

1. Property1 is never
 process p_1 state busy and
 process p_2 state busy;

³ see section 8 for more details.

⁴ some authors use the notation $\forall$ CTL.

2. Set all_processes is $\text{set}(p_1, p_2)$;
 Property2 is
 forall \$procs in all_processes:
 always \$procs.overflow = false;
3. Property3 is
 occurs process p_1 state idle
 after process p_1 event stop_req

Experiences from our case studies show that engineers quickly adopted this style of writing properties. Especially the facility to define parameterised properties (macros) proved to be useful since this allows the engineer to reuse properties.

6.3 Adding Assumptions

Due to abstraction, the verification model normally permits more execution sequences than the real system. Therefore, the model checker sometimes prompts an error trace (counter sequence) to the engineer that is not realistic. In that case the engineer has the option to enrich the property by *assumptions* about the systems behaviour to exclude unrealistic error traces.

Since assumptions usually are only related to some properties and do not apply in general to all properties of the system, the language SPL allows the annotation of assumptions to those properties they are intended for.

As an example for adding assumptions consider property 3 from above. In some rare execution sequences process p_1 never reaches its idle state. Due to abstraction p_1 never becomes active and therefore never gets the chance to perform any actions. We can exclude these traces by formulating the following assumption:

Assumption progress is occurs activeP = p_1 after true

This assumption postulates a fairness constraint⁵ for process p_1 . By adding this assumption to property 3 we restrict the verification model to those execution sequences for which process p_1 always gets the chance to be active after a finite number of execution steps.

Property3 is
 occurs process p_1 state idle
 after process p_1 event stop_req
 with progress

As another example assume we want to verify that after process p_1 has reached the state connect it will send the message release before it reaches its state idle. One possibility to formulate this in SPL is:

Property CheckRelease is
 occurs process p_1 output release

⁵ this is translated to $\text{gf activeP} = p_1$ in LTL.

```

after process  $p_1$  state connect
upto process  $p_1$  state idle

```

The experience of the past years shows that engineers do not like temporal formulae even if they are so simple like the one above. However, they are used to setting flags and testing them somewhere else in their programs. Therefore we support the introduction of history variables (called model variables in SPL). The following SPL property is equivalent to the one above.

```

Assumption AsmRelease is
begin
  dcl MVarWantRelease set(yes,no) init := no
  at process  $p_1$  state connect do MVarWantRelease := yes
  at process  $p_1$  output release do MVarWantRelease := no
end

```

```

Property CheckRelease2 is
  never (process  $p_1$  state idle and MVarWantRelease = yes )
with AsmRelease

```

7 A Survey of the Verification Process

This section is intended to give an overview of the verification process. The following paragraphs should give you an idea how a software engineer interacts with the SVE system during verification. This is accomplished best by looking at the main window of the SVE tool that is shown in figure 3.

First of all the SDL system must be transformed into a system of finite state machines that can be handled by the model checker. This is done by pushing the button **generate a model** in the left column of command buttons. Depending on the size of the SDL system this may take some time. For example, the model generation for one of the GSM case studies (see example 2 in section 8) takes about 6 minutes on a Pentium pro running at 200 MHz.

After model generation has completed the verification model consisting of several BDDs is stored in a file so that it can be reloaded for later verification using button **select a model**. In section 5 we described the automatic model generation in more detail. In contrast to model generation the reloading of a model just takes a few seconds. After the model has been successfully generated the engineer's next task is to formulate some properties in the property language SPL and to store them in an SPL file. In section 6 we described this subject in more detail. In order to check these properties against the verification model one has to load the SPL file using the button **load an SPL file**. During loading the SPL properties are translated into temporal logic and some simple consistency checks are performed, which is usually accomplished in a few seconds.

Fig. 3. main window of the SVE system

Now, everything is prepared for the verification of the properties. Pushing button **verify a property** in the middle column of the main window presents a box, which lists all the properties that are defined in the SPL file. From this box

the engineer starts the verification process by selecting one or several properties at once.

In the lower half of the SVE main window in figure 3 we see the output of a sample session with SVE. After the startup of the system had finished the user selected the already generated verification model GSM and afterwards loaded an SPL file of the same name. Then he started verification of the properties gp1 and gp2. The former property turned out to be false whereas the later property could be proved.

If a property does not hold the model checker starts a diagnosis and computes a counter sequence, which shows why the property does not hold.

A counter sequence is always stored in a file that has the same name as the property but with extension **log**. By pushing button **browse log files** in the right column of the command buttons one can browse through such log files.

Counter sequences are displayed by a special browser that allows the user to manipulate the sequences in various ways. One can apply several filters to highlight or delete portions of the counter sequence in order to enhance its readability. Figure 4 shows the counter sequence for the property gp1.

Since this section is only intended to give a rough idea of the users interaction with the model checker we do not go into the details of the property gp1 and the corresponding counter sequence.

After the engineer has inspected the counter sequence he must decide whether the sequence is a realistic one or not. If the sequence shows a realistic run of the system the engineer found an error in his system, provided the property he checked was vital for the correctness of his system.

If the sequence is not realistic the engineer will either reformulate the property or add an assumption to exclude unrealistic behaviour of the verification model. In section 6 we discussed the topic of properties and assumptions in more detail.

In real life verification it is unlikely that the first check of a property already reveals an error. In most cases the verification of a property consists of several iterations where either the properties are reformulated or assumptions are added.

8 Some Remarks on Complexity

There are certain properties of a system that in general have a major influence on the corresponding state space. When doing multiprocess verification, one important factor is the scheduling of the processes. As we discussed in section 5 in detail, we use a rather restricted deterministic scheduling, which limits process interleaving to a large degree.

The next factor that has a major influence on a system's state space is the use of variables and especially the cardinality of their types. Our experience shows that in most cases the variables could be abstracted either to Boolean variables or to enumeration types with cardinality smaller than 10.

Fig. 4. counter sequence for property gpl

However, the most important factor is the length of the queues. In the worst case, each additional slot in a buffer can result in multiplying the state space by the number of possible messages coming in.

When more messages come in than the buffer can store, the resulting system status is called an *overflow*, normally an erroneous situation. Experiences have shown that if a buffer of short size (about 3 slots) is not large enough to handle all possible states without an overflow, an overflow will also occur when the buffer size is increased. This stems from the fact that one prefers strong handshaking mechanisms in telecommunication protocols.

Therefore, in practice it is normally a good idea to design a system with small buffer lengths in the beginning. If an overflow occurs, one should first investigate how this overflow was caused before increasing the length of the buffer.

At Siemens we are running SVE on a PC environment (Pentium pro 200 MHz with 128 MB RAM). In the following we give some numbers to illustrate the performance we can reach. Of course these numbers do not guarantee that automatic model generation for a system of the same size can be performed within the same time, but certainly they give a rough estimation.

On the other hand, the verification times not only depend on the size of the BDDs involved but also on the type of properties that are checked. However, from a logical point of view the typical properties checked for multi process verification are of almost trivial complexity.

Example 1. Part of layer 3 GSM protocol

Number of processes	3
Reachable state space	8.35×10^7
Time for model generation	3 min
Time for verification of properties	10 s – 30 s
Process 1:	
Number of states	8
Variables	1 variable with type cardinality 8
	1 variable with type cardinality 6
Length of queue	3
Process 2:	
Number of states	9
Variables	no variable
Length of queue	5
Process 3:	
Number of states	2
Variables	3 variable with type cardinality 3
Length of queue	2

Example 2. Part of layer 3 GSM protocol

Number of processes	6
Reachable state space	1.46×10^{13}
Time for model generation	6 min
Time for verification of properties	10 s – 2 min

Process 1:

Number of states	5
Variables	1 variable with type cardinality 2
Length of queue	3

Process 2:

Number of states	6
Variables	1 variable with type cardinality 2
Length of queue	3

Process 3:

Number of states	6
Variables	no variable
Length of queue	2

Process 4:

Number of states	3
Variables	no variable
Length of queue	1

Process 5:

Number of states	2
Variables	no variable
Length of queue	1

Process 6:

Number of states	3
Variables	no variable
Length of queue	1

9 Conclusion

This paper presented the application of model checking to telecommunication protocols for mobile phones. Several case studies have proved that this form of formal verification is learned quickly by engineers and is normally regarded by them as highly capable for finding errors.

So far we have completed a number of case studies to find out if we can apply model checking during our software development in effective and efficient manner. Our main interest at that point was to investigate the capabilities of

formal verification. In these case studies we left our concept of applying verification in the early phases, but post verified the design long after the code was implemented. Therefore we were not too surprised not to find big bugs. But besides its capability of finding errors in system designs, we experienced various other positive side effects when using model checking.

The first one is that a programmer using model checking is forced to think about the system in a different way. The normal way of thinking when writing programs is procedural in terms of program steps like *after step 1 is completed step 2 is executed*. When designing system properties, you have to think in a property oriented way like *always when condition 1 holds condition 2 holds as well*. Most of our engineers applying model checking reported that by thinking about the system in terms of properties they learned a lot about their own programs.

Another benefit in the application of model checking is the possibility to investigate how small modifications in the system design (e.g. priority settings) influence the system properties. Without formal verification the effect of such modifications is hard to determine (there might be one rare case where such a modification causes an unwanted effect). When using formal verification, one just starts the model checker again and it will verify if all system properties specified still hold.

In contrast to other model checking case studies performed at Siemens before, we applied model checking to multi process systems. Although multi process systems are more complex than single process systems the model checker SVE worked efficient enough to handle typical system descriptions of the early design phase.

Our future work will concentrate on extending the supported SDL subset to make the application of formal verification more comfortable for the user.

10 Acknowledgement

We like to thank our colleagues Thomas Filkorn and Peter Bader for their comments about various versions of this paper. Also much credit goes to our colleague Peter Warkentin who coded the initial machine for multi process verification and who also implemented some optimisations for the low level BDD representations of our models. Last but not least we like to thank the referees of this paper for their valuable comments and Bernhard Steffen for his persistency.

References

1. P. Bader, T. Filkorn, and J. Bormann. SPL (SDL Property Language): Syntax und Semantik. Siemens ZT AN 1 (internal document), December 1996.
2. Z.100(1993):CCITT specification and description language (SDL). ITU-T, 1993.
3. E.A. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*, chapter 16, pages 996–1072. Elsevier Science Publisher, 1990.
4. D.E. Long. *Model Checking, Abstraction and Compositional Verification*. PhD thesis, Carnegie Mellon University, 1993.