Verifying Large SDL-Specifications using Model Checking

Natalia Sidorova[1] and Martin Steffen[2]

[1] Department of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, P.O.Box 513,
5612 MB Eindhoven, The Netherlands
`n.sidorova@tue.nl`

[2] Institut für angewandte Mathematik und Informatik
Christian-Albrechts-Universität
Preußerstraße 1–9, D-24105 Kiel, Deutschland
ms@informatik.uni-kiel.de

Abstract. In this paper we propose a methodology for model-checking based verification of large SDL specifications. The methodology is illustrated by a case study of an industrial medium-access protocol for wireless ATM. To cope with the state space explosion, the verification exploits the layered and modular structure of the protocol's SDL specification and proceeds in a bottom-up compositional way. To make a compositional approach feasible in practice, we develop a technique for closing SDL components with a chaotic environment without incurring the state-space penalty of considering all possible combinations of values in the input queues. The compositional arguments are used in combination with abstraction techniques to further reduce the state space of the system. With debugging the system as the prime goal of the verification, we corrected the specification step by step and validated various untimed and time-dependent properties until we built and verified a model of the whole control component of the medium-access protocol. The significance of the case study is in demonstrating that verification tools can handle complex properties of a model as large as shown.

Keywords: SDL model checking; abstraction; compositional, bottom-up verification; verification case study.

1 Introduction

Formal methods, most notably model checking, are increasingly accepted as important part of the software design process [6]. There is a clear tendency to provide validation facilities in the commercial SDL-design tools like ObjectGeode [18] and SDT [22]. Currently, these tools allow to validate SDL specifications by means of exhaustive testing. Due to the high cost of errors in the telecommunication system design, however, complementary ways of debugging and verification are needed. In this paper, we describe the verification methodology we applied to a large industrial software product, namely the control layer of the wireless ATM communication protocol *Mascara* [24].

Formal verification of SDL-specifications via model checking [5] is an area of active investigation [3, 10, 11, 8, 23] (notably, the last two mentioned works are developments of the telecommunication industry itself). Responsible for the increasing acceptance of model checking by industry is its "push-button" appeal, i.e., its promise to allow for fully automatic checking of a program or a system — the model — against a logical specification, typically a formula of some temporal logic. As model checking is based on state-space exploration, the size of a system that can be checked is limited and it is often held that only relatively small systems can be verified with a model checker.

The limitations of model checking by the system size implies that verification is possible only using abstractions and/or compositional techniques. These techniques allow to construct a verification model whose state space is smaller than the one of the original system. However, providing a formal proof of correctness for each abstraction or composition step is prohibitively costly. Aiming primarily at debugging, performing these steps at a semi-formal level does not cause troubles as spotted errors can easily be validated afterwards and checked against the concrete model by the designers and spurious errors can be detected. But in case a property holds for the verification model, one can not claim that the property holds for the system under consideration as well, although the obtained result argues in favour of correctness of the system design. Therefore, we see the primary goal of verification not in proving the overall correctness of the product, but in advanced debugging, finding potential errors in its design and thus increasing its reliability.

For the verification of Mascara, we use the Vires tool-set on the SDL specification, automatically translating the SDL-code into the input language of a discrete-time extension of the well-known *Spin* model-checker. As Mascara is too large to be verified by any existing verifier as a whole, we exploit the protocol's layered structure and perform a bottom-up, compositional verification. In a number of cases, the proved correctness requirements of a component form the basis of its abstraction. This abstraction replaces the real component at the next step when a slice at an upper hierarchical level of the protocol is considered for verification. Doing so we were able to reach the point where the whole control entity of Mascara together with a simple abstraction of the rest of the protocol was taken into account.

The rest of the paper is organised as follows: In Sections 2 and 3 we shortly survey the protocol and the set of design and model check tools we used in the case study. In Section 4 we present the methodology and the techniques applied in the verification, and in Section 5 we highlight results of the investigation. We conclude in Section 6 by evaluating the results and discussing related work.

2 Mascara: a wireless ATM medium-access protocol

Located between the ATM-layer and the physical medium, Mascara is a medium-access layer or, in the context of the ISDN reference model, a transmission convergence sub-layer for wireless ATM communication [1][14] in local area networks. It has been developed within the *WAND*[1] project [24], a joint European initiative by various telecommunication companies to specify and implement a wireless access system for ATM-LANs.

Besides the standard transmission convergence sub-layer tasks such as cell delineation, transmission frame adaptation, header error control, cell-rate decoupling, etc., operating over radio-links, i.e., over a necessarily shared physical medium, adds to the complexity of the protocol. Mascara has to arbitrate *medium access* to the radio environment of a variable number of mobile ATM-stations,[2] provide enhanced error detection and correction mechanisms at various levels to counter the comparatively high bit-error rate of air-borne data-transmission. Last but not least, it has to cater for *mobility* features, allowing a mobile terminal to switch its association with an *access point* in a *handover*.

2.1 Overall structure

From the perspective of verification, Mascara is a large protocol.[3] It is itself composed of various protocol layers and sub-entities (cf. Fig. 1).

The *layer control protocol* together with the *message encapsulation unit* assists in various ways the information exchange between the Mascara layer and entities located within the upper layers. The *segmentation and reassembly* unit does exactly what its name implies: cutting peer-to-peer control messages (also called MPDUs) into ATM-cell size and putting them together upon reception. All three mentioned top-level entities are comparatively unsophisticated and straightforward, as they mainly perform data transformations. The *WDLC*-layer, operating already on cell-level, is reminiscent to conventional (non-ATM) data-link protocols and responsible, per virtual channel, for error- and flow-controlled cell-transmission. The lowest level of Mascara is the *data-pump* including a real-time scheduler, which forms a large portion of the protocol's code-size. Despite its raw size, the functionality offered to the Mascara-layers above is rather simple: the data-pumps of two communicating stations act as duplex, lossy Fifo-buffer. The other large part of Mascara, making up almost half of the SDL-code, is its *control entity,* on which we concentrate here. For a more thorough coverage of Mascara's structure and internals, consult the specification material provided by the Wand consortium [24].

[1] Wireless ATM Network Demonstrator.

[2] Hence the acronym "*M*obile *A*ccess *S*cheme based on *C*ontention *a*nd *R*eservation for *A*TM".

[3] Over 300 pages of (graphical) SDL.

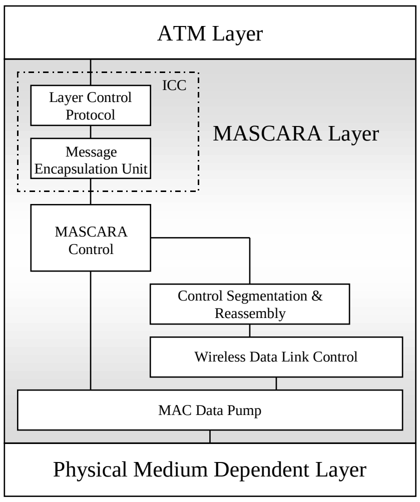
ATM Layer
ICC
Layer Control Protocol
Message Encapsulation Unit
MASCARA Layer
MASCARA Control
Control Segmentation & Reassembly
Wireless Data Link Control
MAC Data Pump
Physical Medium Dependent Layer

Fig. 1. Top-level functional entities

2.2 Mascara control

As the name suggests, the *Mascara control* entity (MCL) is responsible for the protocol's control and signalling tasks. It offers its services to the ATM-layer above while using the services of the underlying segmentation and reassembly entity, the sliding-window entities (WDLC's), and in general the low-layer data-pump.

Being responsible for signalling, MCL maintains and manages *associations* linking access points with mobile terminals, and *connections*, i.e., the basic data and signalling transfer channels, corresponding to ATM virtual channels. Mascara control falls into four sub-entities, each divided in various sub-processes themselves. The two important and complex ones are the *dynamic control* (DC) and the *steady-state control* (SSC). The division of work between the dynamic and the steady-state control is roughly as follows: SSC monitors in various ways current associations and the quality of the radio environment in order to ensure an optimal transmission quality, to keep informed about alternative access points, and to initiate in time change of associations, so-called *handovers.* The dynamic control's task, on the other hand, is to set-up and tear down the associations and connections while managing the related administrative work like address management, resource allocation, etc. Of minor complexity are the *radio control* entity (RCL, with the *radio control manager* RCM as its most important process) and the *generic Mascara control* (GMC).

3 Model checking environment

Dealing with a protocol of Mascara's size, formal validation results with acceptable effort are possible only with appropriate tool support including editing and specification, validation, and of course model checking support.

The tool-set we use for the verification experiments on Mascara is a combination of well-established tools and a number of tools developed within Vires (cf. Fig. 2). Since developing a state-of-the-art model checker from scratch is a daunting task, it was decided to use a powerful model checker as starting point rather than to design a new one. The model checker was enhanced with the ability to deal with time, for Mascara relies heavily on timers.

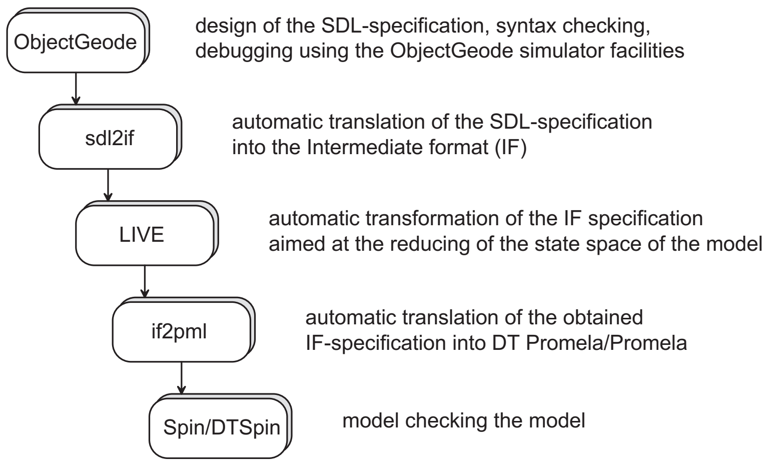
ObjectGeode
design of the SDL-specification, syntax checking, debugging using the ObjectGeode simulator facilities
sdl2if
automatic translation of the SDL-specification into the Intermediate format (IF)
LIVE
automatic transformation of the IF specification aimed at the reducing of the state space of the model
if2pml
automatic translation of the obtained IF-specification into DT Promela/Promela
Spin/DTSpin
model checking the model

Fig. 2. Tool-set for Mascara verification

The tool-set we used especially features:

- ObjectGeode, [18], a Telelogic tool-set for analysis, design, verification, and validation through *simulation*, as well as C/C^{++} code generation and testing of real-time and distributed applications. Targeted especially for telecommunication software and safety-critical systems, ObjectGeode integrates complementary object-oriented and real-time approaches based on SDL [19] and MSCs [17], and recently UML.
- *sdl2if* and *if2pml*, which are the chain of translators rendering SDL into the Intermediate Format If [4], a language for timed asynchronous systems, and If into DT Promela [2] — a discrete-time extension of Promela (the input language of the model checker *Spin*), respectively. Both tools were developed within Vires.
- Live [16], used to optimise If specifications by static-analysis techniques. It transforms an If specification into a semantically equivalent one by adding

systematic resets of non-live variables. The transformation preserves the behaviour while reducing dramatically the global state space (and further, the exploration time). In our experiments, LIVE reduced the state space of the models by a factor of 8 on the average.

– *Spin*, a software package for the specification and verification of concurrent systems [12]. The core of *Spin* is a state-of-the-art enumerative on-the-fly model checker, which can be used to report unreachable code, deadlocks, unspecified receptions, race conditions, and the like. Correctness properties can be specified as system or process invariants (using assertions) or as general *linear-time temporal logic* requirements, either directly in LTL-syntax or indirectly as Büchi automata (called never claims).
– *DTSpin* [2], a discrete-time extension of *Spin*, intended for model checking concurrent systems that depend on timing parameters. It is completely compatible with the standard, untimed version of *Spin*.

4 Methodology

This section describes the methodological aspects of the verification process. The size of the protocol renders any direct, brute-force attempt of model checking out of question. To achieve the main goal, namely debugging the given real-life Mascara protocol, we faced a number of problems, where the most important had been: How to *break-up* the complex program into smaller entities and how to proceed in verifying them? How to *close* the smaller components in order to feed them into the model checker environment? And how to simplify and *abstract* them further in case the components are too large to be accepted by the model checker. We address these questions in turn.

4.1 Bottom-up compositional verification

Our prime goal was to *apply* formal methods, foremost model checking, to industrial protocols, Mascara in this case. With the given overall protocol specification in SDL-92 (Specification and Description Language) [19] as starting point we choose to proceed bottom-up to be able to debug and clean up the single smaller entities with relative ease before proceeding to composed and larger ones. The layered and structured design of Mascara with blocks of processes greatly facilitated this compositional, bottom-up approach to verification.

We started with relatively small blocks of processes from the global specification. First, a model has to be closed by adding an *environment* specification. This environment should be an abstraction of the rest of the protocol. Constructing this abstraction is discussed later. After debugging and verifying a number of properties for simple components, we proceed with considering blocks composed from the verified ones (or their abstractions). Conceptually, the approach corresponds to the rely/guarantee or assumption/commitment paradigm of compositional verification, where the abstractions take the role of the assumptions about the environment.

Using a bottom-up approach in the verification, one gains a lot. Even some magical model checker that allows to feed the whole protocol to it and get the result by just pressing the proverbial button would be of limited use, for it is very well possible, for instance, that some components of the system under consideration are deadlocked, but not the whole system. The model checker tells then that the system is deadlock-free and one should remember to check that no component of the system is deadlocked. The formulation of such a property is not straightforward and involves fairness restrictions and other non-trivial conditions. Going bottom-up, one detects such deadlocks at the very first steps without much effort — the model checker just finds them automatically.

4.2 Closing the model

Sub-models cut out of a global model cannot be verified as stand-alone processes, since they are not self-contained, i.e., the specification of a sub-model relies on the cooperation of the rest of the protocol. It should be noted that Mascara itself, like many other protocols, is an open model in sense that it relies on the existence of an environment whose behaviour is not specified in the protocol. To model-check an open model the user must first transform it into a closed one. Closing models is often performed for exhaustive testing open systems, where processes are introduced within the model to feed it with signal inputs. The way inputs are sent to the model is controlled by these processes and then superfluous or non-significant inputs sequences can be avoided [15, 9].

Adding chaos For the purpose of model checking, the way the model is closed should be well-considered to alleviate the state-space explosion problem: adding even a simple process increases unavoidably the state vector and, worse still, in general the state space. Basically, there are two extreme options how to implement an "outside" environment. One is to construct a simple process behaving chaotically, i.e., sending and receiving arbitrary signals in arbitrary order. In the context of verification of SDL with its asynchronous message-passing communication model, this immediately leads to a combinatorial explosion caused by considering all combinations of messages in the input queues, even if most of them can't be dealt with by the processes and they are discarded. Another option is to tailor the environment process in such a way that it sends the "relevant" signals only, i.e., the ones to which the model under investigation can possibly react. While easing the state-space explosion in the input-queues, it can make the environmental process itself rather complicated and large, multiplying thus the overall state space. At least as detrimental from a practical point of view is that a tailor-made environment requires insight into the model, analysing when and when not it can handle messages. This takes time and is error-prone for large systems such at the components of Mascara.

To avoid both problems, we chose an alternative way: we model the environment as simple chaos but not a separate process external to the model. Instead, the chaotic environment is *embedded* into the model itself by a simple SDL source code transformation. The main idea is quite simple. Since we assume the environment to be chaotic, we must assure that whenever a process is in a state where it can take an input from the environment, it must have a possibility to take this branch. That can be done by replacing this input by the unconditionally enabled None input (thereby abstracting from the sent data at the same time). Outputs to the environment are just removed. For input, the replacement with None effectively removes the (chaotic) data reception from the input action, in this way influencing variable instances in the process appearing as input parameters in those inputs. Therefore, the actions potentially influenced by reception from outside and variable instances whose values consequently cannot be relied on must be eliminated, too. This is done by *data-flow* analysis of the model (cf. [21] for the semantical underpinning of the approach).

Chaotic timers When closing a component, not only all non-deterministic behaviour wrt. to signal exchange must be captured, but also all timed behaviour, which plays a crucial role in telecommunication protocols. The time semantics chosen for Mascara uses discrete-valued timer variables [3]. Ordinary transitions are instantaneous, i.e., they take zero time, and time can progress by incrementing all active timers only when all input queues are empty and there is no None-input enabled.[4]

Now closing the component by adding all possible signal-exchanges renders input from the outside continuously enabled. Especially by incorporating the chaos as sketched above, the branches input-guarded by the newly introduced None-inputs are unconditionally enabled, which means that time may not pass in this situation any longer, for the enabled input actions take priority over the time progress. So due to adding just chaotic sending and receiving of messages, time-outs possible in the original system may not occur after the transformation, in which case time can never pass, a so-called *zero-time loop* occurs. In other words, the simple approach of replacing environment-inputs by None-inputs fails to respect the discrete time semantics of SDL.

In order to preserve the timed behaviour, we must take into account that in any state time the new None-inputs don't forestall potential time-progress. For this purpose, one additional timer is introduced for every process receiving messages from the environment and at every process state, an input from this timer may be taken. This timer takes values Now or Now + 1, where Now means that the timer transition is enabled and messages from the environment may arrive, and Now + 1 means that no messages from the environment will come until the next time slice starts. The decision to set the timer to Now + 1 is taken non-deterministically – the time-out may occur after an arbitrarily many "inputs from the environment". Hence all the behaviour of the original specification is preserved. The pattern of the transformation is shown in Fig. 3.

Another issue concerns the influence of chaotic data received from the outside to the *values* of timers. Like ordinary variables, timer variables can be influenced

[4] More precisely, to allow timer increments, all queues must be empty except saved messages.

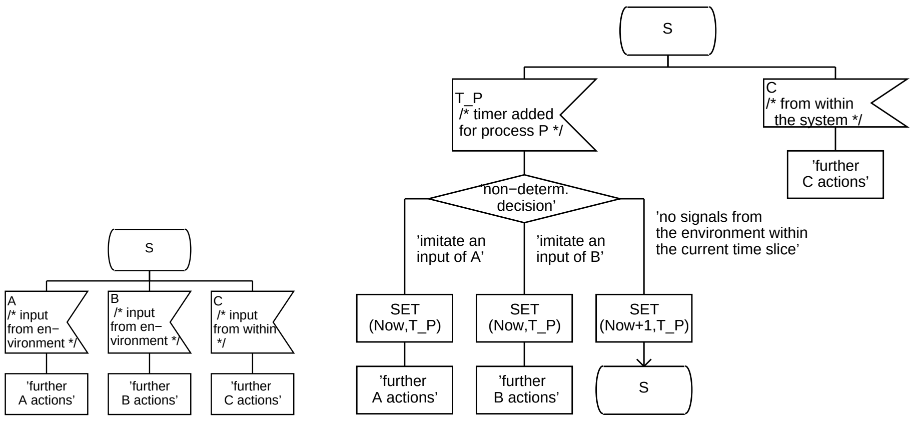

Fig. 3. Transformation of inputs: before (left) and after (right) the transformation

by the reception of chaotic data from outside, but unlike ordinary data variables, we cannot just remove timers whose exact values cannot be relied on. Timers instantiated to an undefined "chaotic" value can expire at an arbitrary moment in time. Therefore, they are treated similar to the ones for inputs from the environment. The operation of setting a timer to an undefined value is transformed into setting it to the $\mathsf{Now} + 1$ value, and correspondent inputs of timer messages are transformed into timer expiration after which a choice is made either to set this timer to $\mathsf{Now} + 1$ and return to the same process state, delaying the timer expiration, or to take the sequence of actions following the actual timer expiration according to the specification. The transformation is shown schematically in Fig. 4.

4.3 Abstraction

One of the main tools of our methodological arsenal was *abstraction.* Abstraction is a rather general technique; intuitively it means replacing one semantical model by an abstract, in general simpler, one. To allow transfer of verification results from the abstract model to the concrete one, both must be related by a safe abstraction relation. The concept of *safe abstraction* is well-developed and has applications in many areas of semantics, program analysis, and verification (cf. [7] for the seminal, original contribution). For *safety properties* in linear-time temporal logic, often paraphrased as "never something bad will happen", the abstract system must at least show all the traces of the concrete one to be used as a safe abstraction. To find safe abstractions of a reactive, parallel system such as a protocol, it is helpful to distinguish between the *data* of a program, i.e., the values stored and transmitted, and its *control*, i.e., the control flow within the processes and their communication behaviour, and, resp., between *data* and *control abstractions.* A third abstraction we routinely used is timer abstraction.

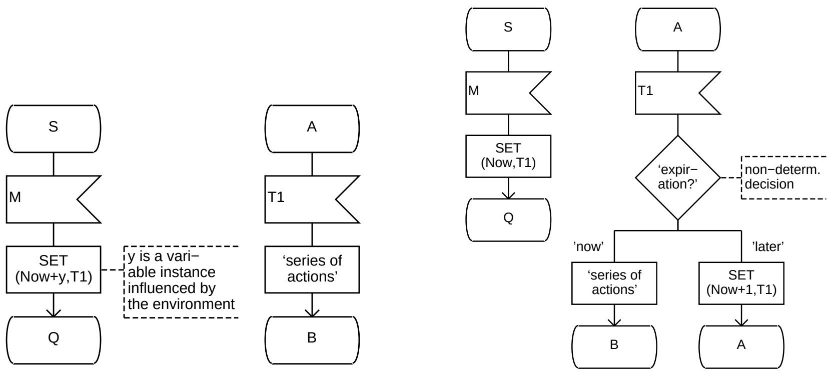

Fig. 4. Transformation of timers: before (left) and after (right) the transformation

Data abstraction Often, the behaviour of a program does not depend on the *specific* values of its data. In this case, many properties of the program stated over the full, often infinite, data domain can be equivalently expressed over finite domains of enough elements. For instance, being interested in a proof that an entity of Mascara handles addresses of mobile terminals correctly and does not give away the same address twice, a two-valued domain of addresses would suffice. This approach is known as *data independence* technique [26].

Control abstraction Given the amount of various entities and processes of the protocol, using data abstraction alone will not yield. The processes of the specification are given in great detail, to serve as the basis for an implementation, and they often possess internally non-obvious behaviour (for instance loops, jumps, conditions depending on data-values, and the like). To deal with this complexity we used a specific type of control abstraction. After a whole-sale entity has being verified against a set of its requirements in the chaotic environment, we replace this entity with an abstraction which was the simplest entity for which this requirements holds.

We illustrate this technique on a simple entity of Mascara, the *radio control* (RCL). Seen from the outside, RCL builds Mascara-control's interface with the lower-layer physical radio modem. Its task is to operate the modem to tune into the terminal with a known frequency upon request, if possible. A property the RCL should guarantee can be phrased as the following simple *response property:*

> "Whenever, after initialisation, the radio control manager receives a request `Acquire_New_AP(newchannel)`, the RCM-process responds either positively or negatively (`Acquire_New_AP_ok` or `Acquire_New_AP_ko`). Moreover, the answer is sent in a given amount of time after getting the request."

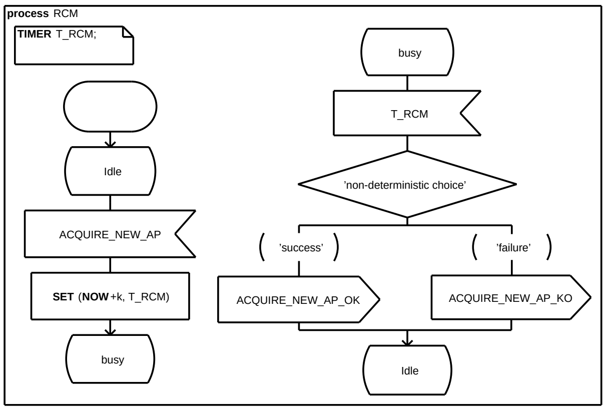

Fig. 5. Abstract radio control manager

The entity must be ready to react upon requests at any time, so it was closed in a *chaotic* environment. To reduce the state space of the verification model, we used data independence limiting the data domain of the parameter `newchannel` with 2 values. We checked the model for absence of zero-time cycles first, afterwards the proper initialisation of the component was checked. Coding the above property in LTL, we could finally verify that the concrete RCL satisfied the property.

Since initialisation of RCL is a confirmed service, and the other entities are initialised only after the initialisation confirmation has been received from radio control, we can abstract away from the initialisation phase in radio control.

After having verified the above LTL-property, one can exploit in the following experiments an abstract variant of RCL which is just one process, radio control manager (Fig. 5). The more sophisticated decisions of the concrete radio control [5] are captured in the manually given, abstract version simply by a non-deterministic choice between a positive or negative decision and the abstraction contains all the information the other components need to in order be verified.

Timer abstraction Another abstraction we apply to cope with the state-space explosion is timer abstraction. A timer whose value is expected to have no influence on the truth of the property can be abstracted by assuming that it can take any value, i.e., it becomes a timer of "chaotic nature" (cf. Section 4.2). Operations on this timer are replaced according to the patterns described for the chaotic timer.

It would seem obvious to verify all non-timed properties with an abstracted-time model and the timed ones with a concrete model, but our experiments show that abstracting the timers may be ambivalent, both with respect to the state

[5] RCL, a small part of Mascara control, takes 9 SDL-pages of the specification.

space and concerning the ability to transfer results from the abstract model to the concrete protocol.

First, the experiments shows that often the state space of the abstracted model is larger than the one of the concrete model when small values for timer delays are taken. In case the behaviour of the protocol strongly depends on timers, abstracting the values of timers may add much behaviour and thus potentially results in a larger state space. But of course, investigating the protocol for various timer settings will require the checking of infinite many combinations of timer settings, and moreover, even when restricting to "representative cases", choosing larger timer setting may in many cases increase the state space beyond the tractable limits. Using just abstract timers, the state space often happened to be manageable for the model checker.

Second, if some functional property is proved with the abstracted-time model, it is shown for all possible values of timers. On the other hand, if the property is disproved or a deadlock in the model is found, the next step is to check whether the erroneous trace given by *Spin* is a real error in the system or a spurious error caused by adding erroneous behaviour either by abstracting from time or by a too abstract environment specification. It can happen that the property fails to hold for the concrete model, however the erroneous trace given by *Spin* is one of the added behaviour. This behaviour cannot be reproduced for the SDL model with SDL simulation tools and we cannot conclude whether the property holds or not. In such a situation one should just redo the experiment using *DTSpin*: one cannot force *Spin* to give the trace from the non-added behaviour, but *DTSpin* guarantees that timers are expiring in the correct order. In our experiments we encountered several cases when using *DTSpin* instead of *Spin*, gave a chance to get a real erroneous trace and disprove the property.

5 Verification results

In this section we shortly survey the verification results. Following the bottom-up, compositional approach sketched above, we obtained a number of results about Mascara control. Starting from MT target cell (MTC, an important part of the steady-state control), we proceeded investigating the steady-state control and the dynamic control, the two largest sub-blocks of Mascara-control (cf. Section 2.2), in isolation, and finally, we verified properties of a model of the whole Mascara control.

Dealing with the various set-ups, we basically follow a bottom-up approach not only proceeding from smaller entities to larger, combined ones, but also advancing from simpler to more complex properties. After a number of reachability checks, we use the built-in Spin features for finding deadlocks and livelocks. The Message Sequence Charts, which are given by Spin and which corresponds to erroneous traces, are analysed on the original model with the help of the ObjectGeode simulator. After correcting discovered structural errors, we proceed to more advanced properties, like *safety*, *liveness*, and *response* properties.

5.1 Reachability checks

Enumerating the whole state space, the *Spin* model checker reports on unreachable code and we use this report as a guideline for formulating reachability properties to check. The report of *Spin* tells which code is unreachable, but it gives no hint why this code is unreachable. Analysing the unreachable code allows to find a reachable point in the specification suspected as the predecessor of an unreachable state. The reachability checks are easily done by just checking *assertion violations* where assertions are inserted at the reachable predecessors of unreachable states. Running *Spin* with an assertion-violation check gives the trace which can be used to look at this reachable state, scrutinising the values of different parameters, states of other processes, etc., to get a clue of what is wrong with the specification. In this way, we found a number of "obviously reachable" states being unreachable and thus a couple of unexpected errors of various kinds.

The reachability checks ensure that the more complicated LTL-properties investigated later are not trivially satisfied.[6] Depending on the entity, typical properties checked were:

– successful/unsuccessful association is possible
– termination of association is possible
– successful connection set-up is possible
– incommunicado cycle is successfully completed.

Used in this way, reachability checking is employed as a sophisticated debugging facility with the assertions used to steer the checker to the critical points of the system. Besides weeding-out errors, we found it likewise very helpful, to use assertion checking (or, a little more complicated, checking LTL-formulas) in a dual way: marking the property of interest as "undesirable" while hoping for its satisfaction — the corresponding "error trace" is useful illustrating characteristic *desired* scenarios. They can be compared with the scenarios provided during the specification phase, thus giving a better understanding of the behaviour of the protocol, and thus enhancing the confidence in the specification.

5.2 Errors Found

Quite a number of errors discovered in Mascara were "just" *programming errors,* including such classics as uninitialised variables (even uninitialised variables due to a typo), forgotten branches in case distinctions, mal-considered limit cases in loops, and the like. Concerning the communication behaviour, we encountered most commonly

– race conditions,
– ambiguous receiver,

[6] Indeed, we started to perform reachability checks regularly after "proving" a sophisticated property only to learn later, that the premise of the implication of this property was unexpectedly false, since unreachable.

- unspecified reception, and
- variables out of range

as general errors at each stage of the verification process. Some of the found error turned out to be false errors caused by the too abstract environment. In this case, the experiment was redone with a more refined version of environment. Reproducing the erroneous trace on the original version of the protocol in the ObjectGeode simulator, those errors confirmed to be real errors in the protocol design, were reported to the developers of Mascara.

Race conditions denote a situation where two signals are sent to an entity "at the same time" such that, due to SDL's asynchronous communication model, the order of reception is undetermined; here we mean more specifically that an unexpected reception order results in an error. Especially prone for this type of error turned out to be the initialisation phases of processes: often, the initialisation signals are given as *unconfirmed* messages. When a number of processes is asynchronously spawned, initialised, and starts communicating under the assumption that the rest of the processes is ready as well, messages may get lost.

Unspecified reception means that a process receives a message in a state where no such message is foreseen; the default reaction in SDL-92 then is to discard the message. The discarding feature is often used on purpose in Mascara's specification, since it saves code, but in some cases the discard is caused by unforeseen behaviour. Given the amount of asynchronous communication activities in the protocol, resulting errors are very hard to detect by code inspection. Signals in the specification with more than one potential receiving process ("ambiguous receiver") also had been a significant source of errors in MCL.

After constructing a small verification model (small compared to the overall specification), we witnessed in several cases state-space explosion without obvious reasons. It turned out that the specification contained some variable that could infinitely decrease or grow. For instance, being informed about deassociation of the same mobile terminal twice — from two different sources — an access point may (under some circumstances) decrease the counter of associated mobile terminals by two instead of one. Thus, the number of associated terminals may become negative. We found it helpful to regularly check that all variables in the model are bounded (their bounds are usually known or can be easily determined).

Besides quite a number of instances of these general errors at each level and besides spurious property violations due to abstraction, errors more specific to Mascara-control model were found and corrected. To exclude "false negatives", each erroneous behaviour was checked against the full SDL-specification by simulation or at least by code inspection and reported to the developers. In the following section, we show one of the more complex properties we verified.

5.3 Time-dependent safety property: unique MAC-addresses

To illustrate up to which extent we could go with the verification, we describe one of the most involved properties verified. It concerns the cooperation of the complete control entity (MT- and AP-side), the interaction of various independently working protocols — notably association handover, the incommunicado protocol, and the "I'm-alive" protocol — and it takes into account the settings of several timers. To maintain an established association between a mobile terminal and an access point, it is important to determine when the association *breaks down* (as opposed to terminating an association properly by deassociating). Driven by various timers, both sides continuously check whether their current association is still functioning.

To determine that an association has gone for good, a mobile terminal and an access point must act independently and rely on their *local* timers, since if the connection is lost, no further communication is possible in the worst case. An important *safety requirement* here is that *"never the access point relinquishes an association before the mobile terminal does"*. This requirement is important for the correct working of Mascara control, especially the correct management of addresses by the dynamic control entity, for if the AP gives up the association, its dynamic control is free to reuse the various addresses allocated to that association for new ones. MT still clambers to reactivate the temporarily broken connection and if it succeeds in doing so, the same addresses will be in use for two different MT's, leading to errors. The property as LTL-formula reads $\Box(\varphi_{mt-lost} \rightarrow \varphi_{ap-lost})$, where proposition $\varphi_{mt-lost}$ describes sending the signal `MT_Lost`, whereby AP's I'm-alive-agent entity gives-up the association. Similarly, $\varphi_{ap-lost}$ captures all situations where the mobile terminal gives up the association by signalling `AP_Lost` or `HO_ind`, both from the MHI-entity.

We established this safety property, if the inequation $\min(\tau_{AP}) > \max(\tau_{MT})$ is satisfied, where τ_{AP} and τ_{MT} are the respective times for the two sides of the association. The two times are bounded according to the following two inequations.

$$\begin{aligned}
\tau_{AP} &\geq (\mathit{Max\_Time\_Periods} + 1) * T_{iaa\_poll} + (\mathit{IAA\_Max} - 1) * T_{frame\_start} \\
\tau_{MT} &\leq (\mathit{Max\_Cellerrors}) * T_{GDP\_period} + (\mathit{Max\_AP\_Index} + 1) * T_{rcm}
\end{aligned}$$

In the inequations, T_{iaa_poll}, T_{frame_start}, T_{GDP_period}, and T_{rcm} are the values of 4 timers determining the behaviour of the above-mentioned protocols, the remaining parameters are program constants of the responsible processes (especially loop bounds). It should be noted that the inequations are not immediate from the SDL-code of MCL: while it is comparatively easy to *identify* the timers that can influence satisfaction of the property by looking at the processes involved, what makes it complicated is the *interference* of the timed reactions: the activities of the various protocols can especially *suspend* other processes temporarily and thus postpone expiration of other timers. With *Spin/DTSpin* it is not possible to automatically derive the equations. Therefore, we verified satisfaction of the safety requirement, resp. checked its violation, for various combinations of values according to the inequations, especially for a number of border-cases, to validate our intuition about the correct interplay of the timers involved.

6 Conclusion

With SDL as the language of choice for the design of telecommunication applications, there is a growing need for formal verification techniques targeted towards SDL and of course corresponding integrated tool support. Currently, most of the work in the field relies on testing and/or validating the design via simulation. For instance in [20], an ATM user-to-network interface is validated using the SDT tool set [22]. The state space is explored by so called bit-state hashing and by random walk traversal. In our work, on the contrary, we use the full state space exploration of the *Spin* model checker, but abstraction techniques instead to deal with the state-space explosion problem. Similarly, [9] explores a number of heuristics or state-saving techniques, especially partial-order techniques, to counter the complexity of state exploration of SDL-specifications, but in the context of simulation. With similar goals and facing similar problems, [13] uses the SDL reachability analyser *Emma* for model-checking telecommunication software. Their tool is based on Petri-nets and it uses (as *Spin* does) partial-order techniques. Unlike our approach, where we rely on the discrete-time semantics as implemented in *DTSpin*, in the work of Husberg and Manner time is modeled by complete non-determinism; so time-dependent properties as the one shown in Section 5.3, cannot be treated. Similarly, the works in [11, 23], also using the *Spin* model-checker, doesn't deal with timing aspects.

A major part of the verification effort expended can be seen as *debugging* the specification. A rightful question then is why to use model checking instead of simulation if model checking is not directly applicable to a large-size model while simulation is. We believe that both methods have their place and well complement each other. Indeed, at the first stage of debugging it is easier and better to use simulation, not model checking. The simple error situations like getting deadlocked already at the initial phase of functioning can be quickly detected by simulation. However, after a number of errors that can be found by simulation are corrected, the model checker shows its strength. For instance, model checker reports about unreachable code which immediately indicates the area of potential problems. Next, the erroneous trace given by a simulator can be very long, and one can not force a simulator to give a shortest one; with a model checker, one can (as most model checkers include a "shortest trail" option). These options significantly simplify the analysis of the cause of an error. Another argument is that only quite a restricted set of temporal properties can be verified via simulation. Model checking enlarge the facilities of debugging in this sense.

One conclusion to draw from our experience of working on the Mascara protocol is that by using state-of-the-art model checking support together with quite a simple methodological approach, one can already achieve a lot. The straightforward approach of using a chaotic closing together with rather simple abstractions has a number of methodological and practical advantages. First, allowing all possible traces by the non-deterministic environment, the safety of the abstraction is immediate. Secondly, closing the model by an environment process takes time; closing it with a more or less chaotic environment can be done fast and routinely. Thirdly, leaving the structure of the entity under investigation untouched allows fast spotting of potential errors, in case the model checker finds a property violation on the abstract level. Moreover, only when retaining the internal process structure it is possible to detect errors concerning the internal loops, conditions, etc., at all. Used in this way, model checking can provide valuable support in increasing the software reliability. As for future work, we expect that the process of verification will greatly benefit from automating some of the routine, but tedious tasks.

References

1. The ATM forum. `http://www.atmforum.com/`, 2000.
2. D. Bošnački and D. Dams. Integrating real time into Spin: A prototype implementation. In *Proceedings of Formal Description Techniques and Protocol Specification, Testing, and Verification (FORTE/PSTV'98).* Kluwer Academic Publishers, 1998.
3. D. Bošnački, D. Dams, L. Holenderski, and N. Sidorova. Verifying SDL in Spin. In *TACAS 2000*, *LNCS 1785.* Springer-Verlag, 2000.
4. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: An intermediate representation and validation environment for timed asynchronous systems. In Wing et al. [25].
5. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
6. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, December 1996.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL, Los Angeles, CA.* ACM, January 1977.
8. A. Barnard F. Regensburger. Formal verification of SDL systems at the Siemens mobile phone department. In *Proceedings of TACAS '98*, *LNCS 1384*, pages 439–455. Springer-Verlag, 1998.
9. J. Grabowski, R. Scheurer, D. Toggweiler, and D. Hogrefe. Dealing with the complexity of state space exploration algorithms. In *Proceedings of the 6th GI/ITG technical meeting on 'Formal Description Techniques for Distributed Systems'.* Universität Erlangen-Nürnberg, 1996.
10. U. Hinkel. Verification of SDL specifications on the basis of stream semantics. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC.* Humboldt-Universität zu Berlin, 1998.
11. G. Holzmann and J. Patti. Validating SDL specifications: an experiment. In Ed Brinksma, editor, *International Workshop on Protocol Specification, Testing and Verification IX (Twente, The Netherlands)*, pages 317–326. North-Holland, 1989. IFIP TC-6 International Workshop.
12. G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
13. N. Husberg and T. Manner. Emma: Developing an industrial reachability analyser for SDL. In Wing et al. [25], pages 642–662.
14. International Telecommunications Union (ITU) , series I recommendations – Integrated services digital networks (ISDN). `http://www.itu.int/itudoc/itu-t/rec/i/index.html`, 2000.
15. Ph. Leblanc. Simulation, verification and validation of models. white paper. `http://www.telelogic.com/download/ObjectGeode/wp_simuv.zip`, February 1998.

16. L. Ghirvu. M. Bozga, J. Cl. Fernandez. State space reduction based on Live. In Agostino Cortesi and Gilberto Filé, editors, *Proceedings of SAS '99, LNCS 1694*. Springer-Verlag, 1999.
17. Message sequence charts (MSC). ITU-TS Recommendation Z.120, 1996.
18. ObjectGeode 4. `http://www.csverilog.com/products/geode.htm`, 2000.
19. Specification and Description Language SDL, blue book. CCITT Recommendation Z.100, 1992.
20. S. M. Shahrier and R. M. Jenevein. SDL specification and verification of a distributed access generic optical network interface for SMDS networks. Technical Report TR-97-18, University of Texas at Austin, Department of Computer Sciences, July 1997.
21. N. Sidova and M. Steffen. Embedding chaos. Technical Report TR-ST-01-2, Lehrstuhl für Software-Technologie, Institut für Informatik und praktische Mathematik, Christian-Albrechts-Universität Kiel, March 2000, submitted for publication.
22. Telelogic Malmö AB. *SDT 3.1 User Guide and SDT 3.1 Reference Manual*, 1997.
23. H. Tuominen. Embedding a dialect of SDL in Promela. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking, Proceedings of 5th and 6th Internaional SPIN Workshops, Trento/Toulouse, LNCS 1680*, pages 245–260. Springer-Verlag, 1999.
24. A wireless ATM network demonstrator (WAND), ACTS project AC085. `http://www.tik.ee.ethz.ch/~wand/`, 1998.
25. J. Wing, J. Woodcock, and J. Davies, editors. *Proceedings of Symposium on Formal Methods (FM 99), LNCS 1708*. Springer-Verlag, September 1999.
26. P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of 13th POPL*, pages 184–193. ACM, January 1986.