Verification of a Wireless ATM Medium-Access Protocol

Natalia Sidorova
Dept. of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, P.O.Box 513
5612 MB Eindhoven, The Netherland
n.sidorova@tue.nl

Martin Steffen
Inst. für angewandte Mathematik und Informatik
Christian-Albrechts-Universität
Preußerstraße 1–9
24105 Kiel, Deutschland
ms@informatik.uni-kiel.de

Abstract

In this paper we report on a model-checking case study of an industrial medium-access protocol for wireless ATM. Since the protocol is too large to be verified by any of existing checkers as a whole, the verification exploits the layered and modular structure of the protocol's SDL specification and proceeds in a bottom-up, compositional way. The compositional arguments are used in combination with abstraction techniques to further reduce the state space of the system. The verification is primarily aimed at debugging the system. After correcting the specification step by step and validating various untimed and time-dependent properties, a model of the whole control component of the medium-access protocol is built and verified. The significance of the case study is in demonstrating that verification tools can handle complex properties of a model as large as shown.

1. Introduction

Model checking [5] is a well-established formal technique for the verification of finite-state systems. 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. In the context of the Vires project[1] [13], the task, however, was to apply model checking to a large industrial software product, namely the control layer of the wireless ATM communication protocol *Mascara* [14].

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, but 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 towards 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.

Our experiments show that by combining relatively simple *abstraction* techniques together with a *compositional, bottom-up* approach, model checking can be successfully applied to large industrial systems. We use the Vires tool-set on the SDL specification of Mascara, 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 for construction of a component 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.

In the verification experiments, we found (and corrected) several errors of various kinds and finally verified a number of behavioural properties including timed ones.

[1] *V*erifying *I*ndustrial *Re*active *S*ystems.

The rest of the paper is organised as follows: Section 2 briefly surveys the protocol, its structure and its tasks. The model checking tool-set we used for the verification is sketched in Section 3. Section 4 describes the methodology we followed. Section 5 briefly presents some of the verification results. We conclude in Section 6 by evaluating the results.

2. The 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] in local area networks. It has been developed within the *WAND*[2] project [14], 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,[3] 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.[4] It is itself composed of various protocol layers and sub-entities (cf. Figure 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 MDPUs) 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 [14].

[2] Wireless ATM Network Demonstrator.

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

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

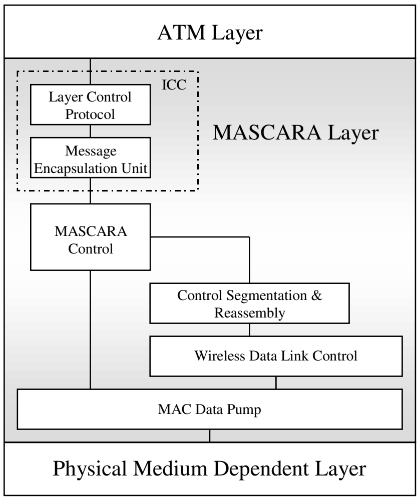

Figure 1. Top-level functional entities

2.2. Mascara control

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

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. Figure 2). The choice was largely determined by the following side-conditions. To facilitate communication with the industrial partners, the specification language was chosen to be SDL. Currently, commercial SDL-tools offer only simulation, but no model checking facilities. Hence, testing is the only way to verify the model with their use. Since developing a state-of-the-art model checker from scratch is a daunting task, it was decided to use a well-established model checker as starting point rather than to design a new one. The model checker was enhanced with adding the ability to deal with time, for Mascara relies heavily on timers.

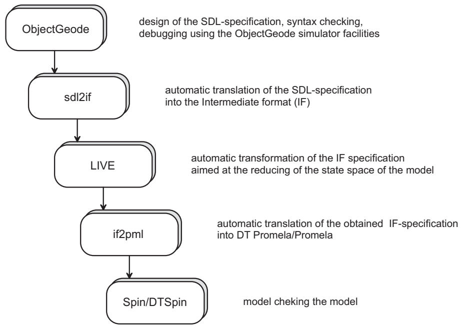

Figure 2. Tool-set for Mascara verification

The tool-set we used especially features:

- OBJECTGEODE, [11], 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 [12] and MSCs [10], and recently UML.

- *sdl2if* and *if2pml*, which are the chain of translators rending 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 [9], 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 [8]. 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

Our prime goal was to *apply* formal methods, foremost model checking, to industrial protocols, Mascara in this case. To be able to do so we needed cooperation with and assistance from our industrial partners, especially *Intracom*. A common email forum, discussing our specification-under-development, a repository keeping track of errors or suspected errors in the specification, versioning of the stages in the specification process, and a number of cooperation meetings with our industrial partner with code walk-throughs, especially in the earlier stages of the Vires-project, were means to obtain an SDL-specification for the verification, as detailed and complete as possible.

4.1. Bottom-up verification

The protocol is given in SDL (Specification and Description Language) [12], a widely accepted, standardised specification language for telecommunication applications. Semantically, SDL is based on processes communicating via asynchronous message passing. To be useful for larger-scale applications, it features definition of data types, allows various ways of grouping and structuring large programs, including object-oriented design principles, and supports a graphical notation. The layered and structured design of Mascara with blocks of processes greatly facilitated the compositional approach to verification.

We started considering relatively small blocks of processes from the global specification. These sub-models, however, can not be verified as stand-alone processes, since they are not self-contained, i.e., the specification of such a sub-model relies on the cooperation of the rest of the protocol. Therefore, it is necessary to *close* a model first by adding one or more *environment* processes specified in SDL at the same hierarchy level as the extracted model itself. 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 assuming some magical model checker, which 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 these deadlocks automatically.

4.2. Abstraction

The size of the protocol rendered any direct, brute-force attempt of model checking out of question, and 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 related with the model of time of *DTSpin*.

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 [15].

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, as a very common type of control abstraction, we simply take a whole-sale entity, such as a process or an SDL-block consisting of a number of processes as it is,[5] and manually abstract away from the rest of the protocol, condensing it into a non-deterministic, *chaotic environment* for this component.

From the methodological point of view, this straightforward approach has three main 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.

Experience with Mascara shows, however, that this simplest approach of a completely chaotic environment is seldom applicable in its pure form, for it causes many spu-

[5] With appropriate data-abstractions.

riously erroneous behaviour, so-called “false negatives”, during model checking. Local livelocks, cycles with non-progressing time, and non-existing deadlocks are typical examples of those false errors. Moreover, the redundant behaviour may also increase the state space. 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 spurious behaviour but it still adds some unwanted behaviour caused by sending non-realistic signal sequences.

These approaches are based not on the knowledge about the behaviour of the rest of the protocol but on the specification of the component under consideration, namely, its input-output behaviour. Both the approaches are safe only if no non-progressing time cycles are added in the abstraction. Otherwise, some behaviour of the system can be lost. On the other hand, the approaches are cheap in sense that such an environment is easy and fast to implement. So they can be considered as a useful kind of heuristics that can be implied at the first stages 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. This process is guided by the understanding of the protocol and the already proved assumptions about the rest of the protocol.

Time abstraction

The closed SDL-model is translated into DT Promela by the translators *sdl2if* and *if2pml*, using LIVE for IF-code optimisation. To run the verification, one has the choice between verifying the timed DT Promela model using *DTSpin* and verification of the model with abstracted time using standard *Spin* (see [3] for the full description of the abstraction of SDL timers we use). It would seem obvious to verify all non-timed properties with an abstracted-time model and all timed properties with a concrete model. However, in some cases it is more convenient to verify non-timed properties with a concrete model as well. If some functional property is proved with the abstracted-time model, it is proved for all possible values of timers. However, 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.

One cannot force *Spin* to give the trace from the non-added behaviour, but *DTSpin* allows to reduce the set of added behaviour by guaranteeing that timers are expiring in the correct order. In our experiments we had several cases when application of *DTSpin*, instead of *Spin*, gave a chance to get a real erroneous trace and disprove the property.

Another argument in favour of timed verification sounds rather unexpected. It is often the case that the state space of a concrete model is smaller than the state space of its abstraction. This can be explained by the fact that the behaviour of the protocol specified in SDL strongly depends on timers. Abstracting their values we add too much behaviour which can result in a larger state space.

4.3. Example

We illustrate the techniques 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.”

The entity must be ready to react upon requests at any time, so it was closed in a *chaotic* environment, with the only restriction that the environment can send only a limited number of signals per time unit. 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 (Figure 3). The more sophisticated decisions of the concrete radio control [6] are captured in the 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 be verified.

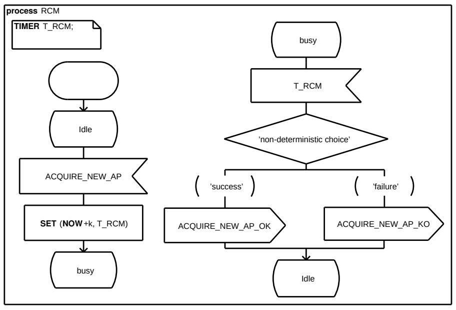

Figure 3. Abstract radio control manager

5. Results

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 including 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. After correcting discovered structural errors, we proceed to more advanced properties, like *safety*, *liveness*, and *response* properties.

5.1. Reachability checks

In all verification configurations, we routinely started with just checking rather simple *reachability* properties, basically check-pointing various crucial steps in the protocols of interest to see whether they are reachable at all. We nevertheless check them, to make sure that the more complicated LTL-properties investigated later are not trivially satisfied. Depending on the entity, typical properties checked in MCL were amongst others:

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

The reachability checks are easiliy done by just checking *assertion violations*. In this way, we found a number of "obviously reachable" states being unreachable and thus a couple of unexpected errors of various kinds. 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-formula) in a dual way: marking the property of interest as "undesired" while hoping for their 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 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,
- unspecified reception, and
- variables out of range

as general errors at each stage of the verification process.

Race conditions denote the 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, and when asynchronously a number of processes are spawned, initialised, and start 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 in such a case is to discard the message. The discarding feature[7] is often used on purpose in Mascara's specification, since it saves code, but in some

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

[7] Mascara is given in SDL-92 [12], the newer SDL-2000 did away with this feature.

cases, the discard is caused by unforeseen behaviour. Given the amount of asynchronous communication activities in the protocol, resulting errors are 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.

In several cases constructing a, compared to the overall specification, small verification model, we got a state-space explosion without obvious reasons. It turned out that the specification contained some variable that could infinitely decrease or grow. For instance, due to being informed about deassociation of the same mobile terminal twice — from two different sources — an access point decreases the counter of associated mobile terminals by two instead of one. Thus, the number of associated terminals can get 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, errors more specific to Mascara-control model were found and corrected. 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 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. 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.

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

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, which 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

Formal methods, most notably model checking, are increasingly accepted as important part of the software design process [6]. Our verification experiments on the non-trivial example of Mascara demonstrates that proceeding the straightforward way we described and using available technology, one can obtain significant results about complex systems. (Though that does not mean that applying model checking for debugging a large software product is an effortless enterprise.)

A major part of the verification effort expended can be seen as *debugging* the specification. A rightful question 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 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. We always started the verification of our models with using the simulation facilities of OBJECTGEODE. However, after a number of errors that can be found by simulation are corrected, the advantages of model checker can be used. For instance, model checker gives a report about unreachable code in the model that 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). That significantly simplifies the following analysis of the cause of an error. One more argument is that only a quite restricted set of temporal properties can be verified via simulation. Model checking enlarge the facilities of debugging in this sense.

One of the minor problems in the effort is, in our experience, finding *properties* to verify. First of all, one can achieve already a lot checking simple properties such as finding dead code and illegal termination. As it stressed before, we found it helpful routinely checking reachability of crucial control-points in the expected behaviours. Moreover, after working on the specification for a while, one gets a fairly good understanding of it, what easily gives scores of properties to check. The functionality of each entity or each group of entities can often be understood as a set of services offered either to a communication peer or to some upper layer, and thus *safety* properties like "each acknowledgment must be caused by a previous request" and *liveness* properties like "each request will eventually lead to an answer". Especially fruitful for finding errors and unexpected reactions are verifying such properties under interference of various protocols.

A clear conclusion to draw is that tools supporting abstractions would extend a lot the applicability of verification. Just with applying LIVE tool with very simple underlying abstraction principles, the state space is in average reduced by one order of magnitude. Some other reduction techniques, which we used *manually* and which, as straightforward as they are, turned out to be effective, could also profit from tool support. Another direction for tool development would be to automatically close the environment, in the simplest case with a chaotic one, or with one reflecting a behaviour defined by a temporal logic formula.

We verified properties of Mascara control as one large entity of the whole Mascara medium access protocol. Debugging the code step by step with enough time and manpower, one could doubtlessly continue in this style repairing more errors and verifying further parts of Mascara. In should be noted, that although the strategy we followed is currently time consuming and tedious, the reasons for it are more of mundane than of theoretical or principal nature. Here, the significance is in demonstrating, that our tools can in principle handle complex properties (including properties depending on timers) of a model as large as shown.

Acknowledgments

The work has been partially supported by the Esprit-LTR project #23498 Vires. We gratefully acknowledge VERILOG for giving us access to their OBJECTGEODE SDL environment. We especially like to thank *Intracom*, who helped in understanding the protocol beyond their duty.

References

[1] The ATM forum. www.atmforum.com/, 2000.

[2] D. Bošnački and D. Dams. Integrating real time into Spin: A prototype implementation. In FORTE/PSTV'98. Kluwer Academic Publisher, 1998.

[3] D. Bošnački, D. Dams, L. Holenderski, and N. Sidorova. Verifying SDL in Spin. In *TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*. 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 *Proceedings of FM 99*, LNCS 1708. Springer-Verlag, 1999.

[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, 1977.

[8] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[9] L. Ghirvu. M. Bozga, J.Cl. Fernandez. State space reduction based on Live. In *Proceedings of SAS'99 (Venetia, Italy)*, September 1999.

[10] Message sequence charts (MSC). ITU-TS Recommendation Z.120, 1996.

[11] ObjectGeode 4. www.csverilog.com/, 2000.

[12] Specification and Description Language SDL, blue book. CCITT Recommendation Z.100, 1992.

[13] Verifying industial reactive systems (VIRES), Esprit long-term research project LTR-23498. radon.ics.ele.tue.nl/~vires/, 1998-2000.

[14] A wireless ATM network demonstrator (WAND), ACTS project AC085. www.tik.ee.ethz.ch/~wand/, 1998.

[15] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In POPL'86, pages 184–193. ACM, January 1986.