computing@tanet.edu.te.ua
www.tanet.edu.te.ua/computing

ISSN 1727-6209
International Scientific Journal of Computing

EXPERIENCE WITH FORMAL VERIFICATION OF SDL PROTOCOLS

Marius Minea [1,2], Cornel Izbaşa, [1,3] Călin Jebelean, [1,2]

[1] Institute e-Austria Timişoara*, Bd. V. Pârvan 4, 300223 Timişoara, Romania
[2] Department of Computing, "Politehnica" University of Timişoara
[3] Department of Computer Science, West University of Timişoara
e-mail: marius@cs.utt.ro, cizbasa@info.uvt.ro, calin@cs.utt.ro

Abstract: *This paper presents a case study in the application of formal methods to the verification of communication protocols. We analyze one component block of telephone switching software developed in the SDL language at Alcatel Network Systems Romania. We use the IF toolset from VERIMAG Grenoble to build a state-transition model of the system and verify selected properties. We present the steps performed for translation and verification and discuss the potential for automating the process and using it on a larger scale.*

Keywords: *formal verification, model checking, communication protocols, specification, SDL*

1. INTRODUCTION

Traditionally, the most commonly used methods for ensuring the correctness of a system have been simulation and testing. While both have their strong points – simulation for evaluating functionality early in the design, and testing for ascertaining the behavior of the actual finished product – they clearly have significant limitations. First, neither simulation nor testing can be exhaustive for any reasonably complex system, leaving open the possibility of unexpected behavior in situations that have not been explored. Moreover, testing takes up a large part of development costs, and errors discovered late in the development process can be prohibitively expensive.

Verification is critical especially for concurrent systems, which often present intricate interactions between components that are difficult to follow and evaluate without formal and automated support. Errors can sometimes occur only for specific execution sequences which are difficult if not impossible to reproduce or debug, making an exhaustive analysis necessary.

Formal verification has matured in the past decade to a point where it provides an effective answer to the above problems. Speaking most generally, it involves building a model of the system under scrutiny, and performing an exhaustive analysis, both model construction and verification being done with rigorous formal techniques. Formal verification is exhaustive, covering all possible system behaviors; it is also highly automatable.

Most major companies in the computer and telecommunication industries have formal verification groups that apply these methods in the design process, and perform in-house research. Moreover, for certain critical systems, the application of formal methods has become a requirement, both for structuring the development process and for verifying the resulting product. For a survey on the state of the art in the field, including numerous industrial examples, see [4].

2. THE VERIFICATION PROBLEM

The SDL language

The Specification and Description Language SDL is supported and standardized by the International Telecommunications Union (ITU-T). It provides both concurrent and real-time aspects and is targeted for the description of communication protocols. Systems are decomposed into blocks and processes, the latter being the unit of concurrency. Code is further modularized into procedures. Processes interact asynchronously via signals that are placed into and consumed from queues. The communication structure is given by signalroutes that connect individual system components.

* This research was partly sponsored by the Austrian Ministries of Education, Science, and Culture (BMBWK), and of Economy and Work (BMWA) under grants GZ 45.527/1-VI/B/7a/2002 and GZ 98.244/1-I/18/02.

SDL has a formal semantics and is thus naturally suited to formal verification. Automatic verification techniques like model checking can be applied by translating the SDL description into an automaton-based representation, which is then exhaustively analyzed by state-space exploration algorithms.

The case study

The code base for the systems developed in SDL at Alcatel is extremely large, both in terms of the number of files and lines of code. The SDL descriptions are complemented by low-level routines written in C. Coding activity involves mostly maintenance and development of new features and integration with the existing system. There is a fairly high amount of unit and non-regression testing. Specifications are natural language descriptions for the individual behavior of the smallest-grained design units, as well as message sequence charts describing the desired scenarios of message propagation through the system.

To assess the feasibility of applying formal verification in this setting, we started out with analyzing a small component part that would be amenable to full state space exploration and also comprehensible for an outsider.

The chosen block serves as an intermediate communication step in the establishment of a telephone connection, in which certain special services are requested from a server. Several types of functionality are possible, and the block is in charge of performing the appropriate type of dialog depending on the parameters of the initiating and subsequent messages.

The block has two interfaces, one upstream to the caller and one downstream to the server, totaling some two dozen signals. Functionality is structured in a single process and 8 procedures, comprising 1500 lines of SDL code (excluding comments).

Most specifications are expressed in terms of unit tests to be performed and involve checking the correct establishment of the dialogue depending on various message parameters. There are also more complex aspects which warrant verification. Messages can carry parameters which are memory references to allocated buffers, which have to be freed properly according to certain rules in the specification. Also, a significant part of the protocol logic is dedicated to checking that both upstream and downstream interfaces are shut down properly when the connection is terminated, which can occur due to a variety of causes.

Related work

Research on verification of SDL designs has been done both in academic and industrial settings. At Siemens, the verification of a layer of the GSM protocol is described in [8]. The in-house BDD-based model checker SVE is used to verify designs of up to 6 processes and reachable states.

The developers of the IF toolset used in this paper have performed several case studies [3], including a standardized protocol (SSCOP) developed by France Telecom, with 2000 lines of SDL code, and the control layer of the MASCARA ATM protocol with 3000 lines of SDL code [7]. The latter study especially contains a good quantitative comparison of the reduction benefits obtained by various methods, including live variable analysis, partial order reduction, and compositional reasoning. The MASCARA protocol was also analyzed using the Spin model checker using a translation from IF as a starting point [1,9].

3. VERIFICATION FLOW

Our goal was to use existing verification tools as much as possible, identify pluses as well as limits and draw conclusions about the most needed developments that would allow large-scale applicability to industrial-size systems.

While there has been some considerable interest for the verification of SDL designs, there are few freely available systems that deal with SDL code directly. Some verification tools based on Petri nets have been built with the intention of accommodating SDL as an input language, but currently SDL support is either restricted to a nonstandard subset (PEPtool from the University of Oldenburg) or not yet complete (the Maria tool from Helsinki University of Technology).

We have selected the IF toolset [2] from the VERIMAG research laboratory in Grenoble. IF is an intermediate representation and validation environment for timed asynchronous systems. It consists of a textual language which is suitably expressive for the description of a large class of such systems, translators that interface with commonly used description languages such as LOTOS and SDL, and a toolset that supports state space exploration, simulation and validation.

Code translation

As a first step, the SDL description was translated into IF. We used the `sdl2if` translator, which is part of the IF toolset, and relies on the SDL API provided by the Telelogic ObjectGEODE SDL compiler to access the intermediate representation built during parsing.

Since IF was designed largely with the translation of SDL in mind, the result is structurally very similar to the original SDL description, but reduced to a set of primitive language features suitable for analysis. Thus, working with the IF translation should be easy for a designer familiar with SDL if the process were to be used in a larger setting.

One limitation of the current version of the `sdl2if` translator is that procedures are handled through inline expansion. This resulted in a code blowup from 1500 lines of original SDL code to 26000 lines of IF representation. While the resulting IF code was still analyzable within reasonable performance limits, it resulted in a high overhead in generating, parsing and compiling the model. Moreover, the resulting code was difficult to follow visually, something which was important at this experimentation stage, especially since the verification team was not familiar in detail with the code functionality.

Consequently, we separated the translation of each procedure. Since the procedure call graph for this example is acyclic, only one instance of each procedure can be active at any moment. Thus, it suffices to introduce a "return address" variable of enumeration type for each procedure, which identifies the call location in the program text, and is tested upon exit from the procedure to determine the proper next state. Procedures which were called only once were left in their inlined expansion. This straightforward change reduced the generated IF code to 2000 lines.

Other code processing

Further treatment was necessary due to the semi-formal modeling style adopted, which mixes SDL with code fragments written in C. SDL allows both decisions (conditionals) and tasks (such as assignments) to be specified as arbitrary text strings, which are uninterpreted. The outcome of a decision in such a case is specified as being unknown. In our case, conforming to a fairly typical modeling style, the descriptive strings are followed by comments which contain the actual C code for the decisions and tasks respectively. This code is handled appropriately by the Telelogic ObjectGEODE suite when compiling SDL into C. Using scripts, we embedded the C actions into the generated IF output and translated them into IF syntax.

Another related issue was the preprocessing which is performed on message sends and receives. The SDL portion of the code contained signals with no parameters. In reality, messages sent and received have a large number of fields. These are handled by preprocessing routines written in C that copy the relevant message fields to and from appropriate variables in the context of the process. For the purpose of this study, the appropriate signal parameters and code stubs on send or receive were inserted into the IF code by hand. For large-scale applicability, automation of this procedure is feasible considering that the corresponding code is clearly structured and marked for treatment by the SDL compiler.

Abstraction

At this step, performing an abstraction on the resulting design becomes necessary. As already mentioned, the messages sent and received by the block under consideration have a rather large number of fields. Some do not directly influence the behavior of this block, but are either forwarded from an interface to another, or are set by the block for use by the communicating blocks upstream or downstream. Their relevance to the property under verification can be determined automatically by a dependence analysis (live variable analysis or program slicing). The IF toolset incorporates a source-to-source translator `if2if` that performs such simplifications on the design. For the studied example, since message fields and their treatment had to be inserted by hand from the C description, only the fields actually used in the SDL portion of the code were inserted in the first place.

In addition, the data width of some variables was reduced in order to cut down on the state space of the system. The protocol relies on memory references to buffers being transmitted in certain message fields. From the point of view of the block under consideration, references are an abstract data type for which the only operations are copying and testing for a null value. Thus it is formally justifiable to reduce the width of reference variables to one bit.

After this processing, the resulting model has 50 control states (out of which 13 are stable and the remainder are introduced in the translation to model decisions and procedure returns), and 25 bits of data, which accounts for a potential state space of about 400 million states.

Verification Interface

The `if.open` compiler translates the IF description into a C source file which contains data structures and functions for representing and exploring the corresponding transition system. This C description can be linked with the toolset's libraries to produce one of several standalone executables for analyzing the system behavior. One such program is a simulator which provides a text interface presenting at each point the possible transitions that can be taken by the system, one for each message (with various values for the parameters) produced by the environment. It is also possible to produce a state space generator which writes out a textual description of the model as a labeled transition system for subsequent use by other tools, and an evaluator for specifications written in alternation-free μ-calculus.

Interfaces with other verification tools

Through the labeled transition system produced by the generator, the toolkit can interface with the CADP (Caesar-Aldebaran) verification toolset [5] developed at INRIA. This allows several processing steps to be done on the automaton description of the system, as well as the use of alternative verification algorithms. An useful such step is the renaming of transition labels in the model description, for instance in order to collapse transitions with the same message but different parameters. Another option is the hiding of transitions, which is useful when the specification refers only to several relevant transition, the rest being ignored. CADP supports minimization of automata after performing such transformations as well as bisimilarity checking of two automata, and μ-calculus formula evaluation.

Using another translator, `if2pml` [1], developed at Eindhoven University of Technology, IF can interface with one of the most widely used model checkers, Spin [6], written at Bell Laboratories. Spin is also targeted specifically towards communication protocols, with an input language (Promela) resembling both C and Communicating Sequential Processes. In addition to verifying specifications written in linear temporal logic (LTL), Spin incorporates an interactive simulator which can represent a system run using Message Sequence Charts. This is particularly useful, since the functionality of the block under analysis is also described using MSCs, a common formalism for the specification of communication protocols.

Most of the verification results we present have been obtained with the IF toolset directly. We are presently evaluating both CADP and Spin as alternatives to complete the study.

4. VERIFICATION RESULTS

Lightweight Verification

As might be expected, we have found both the simulator facility in the IF toolset as well as the simulator in Spin useful for validating the functionality of the protocol and our understanding of it, and for presenting our results to the designers. While not being an aspect of formal verification proper, such complementary support is crucial for developing confidence in the verification results.

The simplest kinds of specifications for the model under analysis are those used for unit testing. Most of the time, they specify that the reception of a particular message is followed by the emission of another message, perhaps with some correspondence between their parameters. Moreover, emission is usually an immediate consequence of the reception.

For these cases, one particular feature of the IF toolset has proved very useful. The generator which produces the textual description of the state-transition system labels a transition with the message received or sent on that transition, or with "i" for an internal transition. Optionally, the generator can also produce the unstable states introduced in the translation to handle decision points, in addition to the stable states. This option results in transitions that correspond to one reactive step of receiving a message, performing some computation and sending the response. In this case, the transition label is a concatenation of messages received and sent, with their parameters. A sample label might read:

-(proc,q,mrcv,{{0,0}}) +(proc,q,msnd,{{0,0}}) i i

For unit tests where the output is emitted in the same transition upon receiving the input, this provides a very simple check. We generated the labeled state-transition system and extracted all transition labels (about 300). For all input/response unit tests we then checked whether the list contained precisely the expected message matches.

With this test, we found a potential problem in the release of buffer references sent as message parameters. The specification states that for any message whose references are not retransmitted on the other interface, the references have to be freed (by emitting appropriate messages for this task). We identified that for one particular closure message, the references were not freed in some states and reported the issue to the designers. Detailed analysis of the scenario revealed this not to be an error, since no valid references are expected in that case.

Checking μ-calculus formulae

The evaluator built by the IF toolset can check temporal formulas written in an alternation-free fragment of the μ-calculus. This provides an expressive means of specifying a wide variety of properties.

In brief, formulas are evaluated as sets of states; a specification is true if it is satisfied by the initial state. If f denotes a formula, and *a* denotes an action, the temporal modalities are:

`[a]f` $= \{ p \mid \forall q \,.\, p \xrightarrow{a} q \Rightarrow q \models f \}$

(the states for which all *a*-successors satisfy f)

`<a>f` $= \{ p \mid \exists q \,.\, p \xrightarrow{a} q \wedge q \models f \}$

(the states with an *a*-successor that satisfies f)

It is possible to use sets of actions, separated by denoting "or", their complement, denoted by ! or the set of all actions, denoted *. Using the least and greatest fixpoints as primitives, the following useful modalities can be defined:

```
all f = gfp X.(f and [*]X)
pot f = lfp X.(f or <*>X)
inev f = lfp X.(f or [*]X and <*>T)
```

A first and natural property to verify is absence of deadlock. This is specified by all<*>T (meaning that in every state some transition is possible).

We continued by analyzing several properties for which unit tests are specified. Their general structure is the following: assuming that a certain sequence of messages – perhaps with some given parameters –

occurs, this sequence will be followed by one specified message. For instance, if an opening message "open" is answered by a connection establishment "est" and then by an end of selection "isel" from downstream, the block will respond with an end of selection "osel" on the upstream interface. Since the system has one "silent" transition before waiting for the initial message, this can be written:

[*]<"-open">pot<"-est">pot<"-isel">inev["+osel"]T

This property and a few similar ones are verified practically instantaneously.

A further property states that in a certain state, a signal signifying progress of the call has to be accepted without influence on subsequent behavior. For this, we verified a necessary condition, that the set of states in which the progress signal can be accepted once is the same as the set of states in which the progress signal can be accepted twice:

all (<" -prog">T <=> <" -prog"><" -prog">T)

If a message containing an invalid reference is sent, the block will be informed of this error by a message that requests closing down the current dialogue. A requirement specifically stated in the specification is that this message has to be accepted at all times. To specify this property, we need to identify and express the set of stable states, where the block can accept messages, as opposed to the intermediate transient states where other decisions are taking place. To this effect, we first renamed all message outputs to internal transitions. The desired stable states are then those for which some other transition other than internal transitions or passage of time (tick) are possible, and the property is stated:

[" i"]all(not["! i | tick(-1)"]F => <" -abort">T)

Finally, a property which was deemed particularly interesting by the protocol designers is that each of the two protocol interfaces shuts down properly. The protocol logic implements this by sending a closure request "cls" and waiting for a confirmation; in case of a timeout, closure is forced by sending the confirmation "cls" directly. This dialogue is not needed if the block receives an abort message with a parameter specifying that the interface is already closed. This means verifying that the stop state cannot be reached without one of these messages occurring, i.e., the following is false:

lfp X.<"!+cls|-cls|-abort(closed)">X or ["-tick(-1)"]F

The system compiled from the IF description after performing the handling of procedures described in the previous section is reduced from 875000 states (for inline expansion of procedures) to 140000 states. This can be further reduced to 30000 stable states if the transient states are collapsed when generating the labeled transition system.

Individual time and memory consumption for the above properties, on a Pentium III machine at 750MHz is given in the following table:

Table 1. Verification performance

Property	Time (s)	Memory (MB)
no deadlock	50	27
progress	67	36
abort	53	36
closure	16	19

5. CONCLUSIONS AND FUTURE WORK

As a first positive outcome, we have achieved our goal of showing that formal modeling and verification is applicable and useful for the system that we set out to analyze. While the result is not push-button verification, and a significant one-time effort was involved in performing the study, many of the processing tasks that we performed manually are straightforwardly automatable, and a repeat effort will be significantly faster to carry out.

With respect to the properties analyzed, those corresponding to unit tests are verified almost instantaneously. Also, their specifications, sequential in nature, are far simpler to write than temporal formulas in their full generality and perhaps with the aid of pattern libraries -- should be easily accessible to designers. We were favorably surprised by the amount of information that can be extracted just from the labels of the transition system and consider this "lightweight" analysis to be valuable as an initial check.

The full value of an exhaustive formal analysis becomes apparent for more complex global properties. While we identified and verified several such properties (e.g., reference treatment and proper closure) even in the case of this single block, such properties will occur more often when verifying larger subsystems.

In terms of code size, our study is comparable to state-of-the-art studies performed so far, though probably less complex in structure. The resources used are still far from the limits of the currently available computing platform, which should allow scalability to multiple components larger in size.

One aspect of future work concerns automation, especially related to the identified modeling style, with action descriptions and external routines written in C code. A second, more fundamental aspect concerns scalability. A promising approach is by hiding of selected transitions and subsequent minimization, as employed in the CADP toolset. We will attempt to perform simplification already at the source level, to avoid generating complex systems in the first place. We would also like to investigate compositional reasoning, by defining interfaces for each component, at appropriate levels of abstraction. All of these should bring us closer to the goal of handling end-to-end properties of large systems.

Acknowledgments: We thank Marius Bozga of VERIMAG for his time and patience in providing support for the `sdl2if` translator and answering questions about the IF toolset. At Alcatel, special thanks go to Florin Bunescu who as leader of one of the design groups provided constant help with everything we needed to carry out the study, and to Delia Golcea who as software development manager supported this project with great interest from the very beginning. We also thank Telelogic AB for providing a trial license of their SDL API interface.

6. REFERENCES

[1] D. Bošnacki, D. Dams, L. Holenderski, N. Sidorova. Model checking SDL with Spin. *Proceedings, 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, Springer Verlag, LNCS 1785, pp. 363–377.

[2] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, L. Mounier. IF: A validation environment for timed asynchronous systems. *Proceedings, 12th International Conference on Computer Aided Verification (CAV 2000)*, Springer Verlag, LNCS 1855, pp. 543–547.

[3] M. Bozga, S. Graf, L. Mounier. Automated validation of distributed software using the IF environment. *Proceedings, Workshop on Software Model Checking*, Elsevier Electronic Notes in Theoretical Computer Science 55(3) (2001).

[4] E.M. Clarke, J.M. Wing. Formal methods. State of the art and future directions. *ACM Computing Surveys*, 28(4) (1996), pp. 626–643.

[5] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, M. Sighireanu. CADP: a protocol validation and verification toolbox. *Proceedings, 8th International Conference on Computer Aided Verification (CAV'96)*, Springer Verlag, LNCS 1102, pp. 437–440.

[6] G. J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering,* 23(5) (1997), pp. 279–295.

[7] G. Jia, S. Graf. Verification experiments on the MASCARA protocol. *Proceedings,8th International Workshop on Model Checking of Software (SPIN 2001)*, Springer Verlag, LNCS 2057, pp. 123–142.

[8] F. Regensburger, A. Barnard. Formal verification of SDL systems at the Siemens mobile phone department. *Proceedings, 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, Springer Verlag, LNCS 1384, pp. 439–455.

[9] N. Sidorova, M. Steffen. Verifying large SDL-specifications using model checking. *Proceedings, Meeting UML: 10th International SDL Forum (SDL 2001)*, Springer Verlag, LNCS 2078, pp. 403–416.