



Model Checking Software with Well-Defined APIs: the Socket Case

P. de la Cámara, M.M. Gallardo, P. Merino and D. Sanán *

*Dpto. de Lenguajes y Ciencias de la Computación University of Málaga, 29071 Málaga, Spain

1 Introduction

Model checking is an automatic verification technique which basically consists of exploring every possible behavior of a model (a simplified version) of a system, in order to prove whether it fulfills a set of critical properties. As is well-known, the main drawback of model checking is the state explosion problem which occurs when the model being inspected requires more resources than are available. Clearly, the ideal objective of model checking techniques is construct tools that directly analyze the final software, thereby avoiding the use of intermediate models. There exist several proposals in that direction such as Bandera [CDH⁺00], SLAM [BCLR04] and JPF [VHBP00] which can perform the so-called software model checking. This paper is a summary of [dlCGMS05] where we described a different approach to analyzing final software. Our proposal is based on the “model-extraction” technique that consists of *automatically* translating the program to be analyzed into a model, described using a “formal description technique” (FDT) used as input of an existing model checker. Translation implies a reduction (abstraction) of some details of the original program in such a way that the resulting model only contains the relevant aspects for a given set of properties. A common problem in model checking software is the presence of calls to external functions (XFC) inside the analyzed code. For instance, many applications are written in C/C++ code using calls to the socket API functions. In order to deal with these functions, we have modelled their behaviour by means of an FDT. Thus, by selectively removing the XFC and replacing them with their corresponding models, we may construct closed models fully described with an FDT. We have implemented our proposal in the tool SocketMC which is able to construct a PROMELA model, to be analyzed with the SPIN 4 model checker, from a C code including calls to the socket API.

2 Modelling the Socket API

The translation into PROMELA of a particular C system with calls to the socket API must take into account three different aspects: the control flow, variables and data, and the translation of the API calls.

Loops, selections and jumps in the original C code are just replaced with the `do`, `if` and `goto` PROMELA statements. In addition, C variables are preserved in the model. Every process in the new PROMELA model contains a state structure with all the C variables being referenced as fields in this

(SOCKET)	$e \xrightarrow{\text{socket}_i(v)}_s \langle \dots, e_i[k/v], \dots, e_{ep}[\langle v, cre, \perp, \perp, \perp \rangle / k] \rangle$
(BIND)	$e \xrightarrow{\text{bind}_i(v,p)}_s \langle \dots, e_{ep}[\langle v, bou, p, \perp, \perp \rangle / k] \rangle$
(CON-ACC)	$e \xrightarrow{\text{con}_j(v,p) \parallel \text{acc}_i(w,w'')} \langle \dots, e_i[k''/w''], \dots, e_{ep}[\langle w'', con, p, b, k' \rangle / k'', \langle v, con, p', b', k'' \rangle / k'] \rangle$
(API1)	$\langle \dots \parallel c_i \parallel \dots, e \rangle \xrightarrow{I(c_i)_i \in \text{scall}_i, e \xrightarrow{I(c_i)_i} e'} \langle \dots \parallel \text{next}(c_i) \parallel \dots, e' \rangle$

Figure 1: Excerpt of semantic rules for SCALL and interleaved process execution

structure. This method allows us to use auxiliary C functions and macros (i.e. `inet_aton` for manipulating `sockaddr_in` structures) preserving the behaviour of the original system. Furthermore, it is necessary to declare additional variables which are useful in the underlying implementation of the socket API modelled and in the calls to the operative system (OS).

In the new model constructed, the external calls must also be transformed. For example, a call to `open` from the socket interface is translated into a PROMELA macro. This macro contains embedded C code that implements the API functions and the OS behaviour.

In order to prove the correctness of the transformation proposed, we give a formal operational semantics of the original API ensuring that our implementation matches the semantics. Figure 1 shows an excerpt of the rules defining the semantics of the API socket (\rightarrow_s). These rules formalize single execution steps in the C processes, showing how the execution of each API call modifies the so-called environment, that is, the values of the process variables. For example, rule CON-ACC defines how to create a TCP connection between two *end_points*. Each *end_point* is defined as tuple $\langle name, state, port, buffer, pair \rangle$. A pair of *end_points* compose a *stream* which is a reliable bidirectional communicating channel between two C processes.

We also formalize the interleaving of processes in execution (rules for \longrightarrow). To this end, we define a configuration as a pair $\langle c_1 \parallel \dots \parallel c_n, e \rangle$, where each c_i represents the program counter of the C process i . Thus, rule API1 defines how the socket functions, except `connect` and `accept`, modify configurations.

As commented above, XFCs must be replaced by a model of the corresponding function described in a TDF. Thus, given a formal description of the API, we build a model for the original C program combining PROMELA and C. Our design pattern consists of describing blocking and non-deterministic behaviours by means of PROMELA, and using C for the manipulation of variables. We need a set of global variables in order to represent the OS state. Since both these variables and the model for XFCs have been specifically designed for model checking tasks, the amount of memory used for OS representation is quite reasonable. It is necessary to recall that the model checking techniques are limited

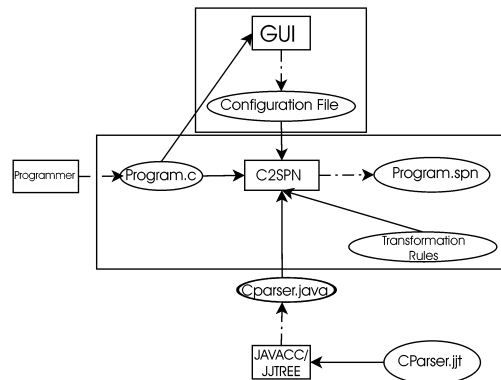


Figure 2: Overall tool architecture

by the amount of memory necessary to represent all the possible states of the system. Although the socket API handles very large message buffers, real programs are only able to understand a limited set of messages. Consequently, we represent the messages by means of unique identifiers and store actual message contents outside the state vector. In our model, buffers consist of a compact queue of identifiers; so, when programs need to read a message, the modelled XFCs recover the message content using its stored identifier. Another important feature of the API socket is the use of blocking sockets. Some XFCs block the execution of the calling process until a condition is met. We use PROMELA guards in order to guarantee that the modelled XFCs will block under the same circumstances. In addition, real socket connections may fail for numerous reasons. A complete model should take these possible failures into account. For this reason, some XFCs display different behaviours representing both the correct execution of the operation and some anomalous behaviours of both OS and the network. Again, we use PROMELA to express the non-deterministic selection of one of these behaviours.

We have implemented some new reduction techniques that may be combined with those already incorporated in SPIN to deal with the state explosion problem. For instance, OS message buffers are compressed using dynamic identifiers and tables with actual values. This compression technique is also applied to other important OS data structures (IP addresses, port, etc). Furthermore, it is possible to take advantage of inherent symmetries in client-server applications, using symmetry reduction; for instance, in scenarios where several equal client processes connect to a single sever. Symmetry reduction correctly analyzes a single interleaving in the clients from among all possible executions, which considerably reduces the number of reachable states.

3 The tool SocketMC

The architecture of SocketMC is described in Figure 2. The conversion from C programs to PROMELA involves two steps. First, a graphical application allows the user to add files containing the C implementation of the processes and gives him/her the opportunity to specify which functions shall be treated



as XFC. The GUI also provides the configuration file and generates specific C code to every process. The translation from C into PROMELA is performed by a Java application called C2SPIN. C2SPIN takes as input all the C files specified in the configuration file, those generated by the GUI, and the file containing the transformation rules. The final PROMELA model is generated by applying the transformation rules for XFCs and the language. The transformation rules describe how to deal with every XFC. Thus XFC can be transformed according to the specification of the model or it can even be eliminated if the corresponding call need not be analyzed.

In future versions, we are thinking of integrating the tool SocketMC with α SPIN [GMMP04]. Thus, we will be able to reduce the state explosion problem adding the abstraction technique implemented in α SPIN to SocketMC.

References

- [BCLR04] T. Ball, B. Cook, V. Levin, and S.K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. *IFM*, 2004.
- [CDH⁺00] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby S. Laubach, and H. Zheng. Banderas: Extracting finitestate models from java source code. ACM Press, 2000.
- [dlCGMS05] P. de-la Cámara, M.M. Gallardo, P. Merino, and D. Sanán. Model Checking Software with Well-Defined APIs: The Socket Case. In T. Margaria and M. Massink, editors, *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, pages 117–126. ACM-Sigsoft, 2005.
- [GMMP04] M.M. Gallardo, J. Martínez, P. Merino, and E. Pimentel. α spin: a tool for abstract model checking. *International Journal on Software Tool for Technology Transfer*, 5:165–184, 2004.
- [VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *IEEE Computer Society*, pages 3–12, Grenoble, France, sep 2000.