Towards Model Checking C Code with OPEN/CÆSAR [*]

María del Mar Gallardo, Pedro Merino and David Sanán

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

Abstract. Verification technologies, like model checking, have obtained great success in the context of formal description techniques (FDTs), however there is still a lack of tools for applying the same approach to real programming languages. One promising approach in this second scenario is the reuse of well known and stable software architectures originally designed for FDTs, like OPEN/CÆSAR. OPEN/CÆSAR is based on a core notation for Labeled Transitions Systems and contains several modules that can help users to implement tasks such as reachability analysis, bisimulation, and test generation. All these functions are accessible with a standard API that makes it possible the generation of specific model checkers for new languages. In this paper, we discuss how to construct a model checker for C distributed applications using OPEN/CÆSAR.

1 Introduction

The difficulty of constructing reliable complex software is well known, especially when developing distributed communication systems. Formal techniques such as *model checking* help us to improve the quality of these systems, assuring the satisfaction of certain critical properties (typically temporal properties). Traditional model checking tools (SPIN[5], CAESAR[6]) have been oriented towards analyzing system models described in a particular high level language also known as formal description technique (FDT). For instance, the modelling language PROMELA is the input for SPIN, while process algebras are valid inputs for CAESAR. However, currently, many academic and commercial projects are focused on extending the techniques and algorithms developed for FDTs to the usual and more complex implementation languages. This is the case of Bandera and JPF [4] for JAVA and CMC and SOCKETMC [2] for C.

There exist two main approaches to attack the problem of software verification. On the one hand, Feaver, Bandera and SOCKETMC, translate the original system, written in a programming language, into a particular FDT that is the standard input of an existing model checker. This method, that is called "model-extraction", allows us to reuse the target tool, but having into account the additional effort of constructing a high level model from the original system.

The second approach to verify software consists in *implementing* new "language-specific tools". Clearly, this method may involve a considerable amount of development

[*] Work partially supported by projects TIN2004-7943-C04-01 and TIC2005-09405-C02-01.

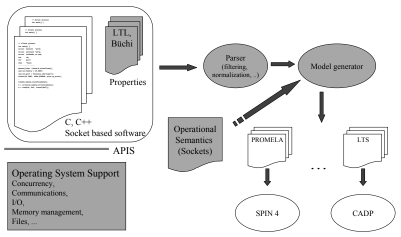

Fig. 1. Schema for verifying systems with well-defined APIs.

work. Fortunately, some frameworks may assist in this task. For instance, the toolset OPEN/CÆSAR[3] permits the construction of specific model checkers, having as input models described using a labeled transition system (LTS). The tool also provides an application programmer interface (API) which facilitates the construction of LTSs and diverse libraries which, for instance, provide structures to store the states or compute hashing functions.

In this paper, we propose to extend the range of input languages for the OPEN/CÆSAR framework, adding the possibility of verifying the popular C language. Moreover, and following the idea of SOCKETMC, the tool will be able to verify client_server applications that make an extensive use of an API, e.g. the Socket API. Figure 1 gives an overview of the process followed to verify systems that use well-defined API. As an initial step in the construction of a specific model checker, this paper describes how to translate client_server applications into the data structures employed by OPEN/CÆSAR. The new representation obtained will allow us to have more control over the algorithms and data structures which play a crucial role in the model checking process. For instance, we may add new features to the model checker such as variable compression, hashing, abstraction, and the localization of errors in the original code.

2 Integrating C into OPEN/CÆSAR

In this section, we describe the process of constructing LTS representations from C programs. This transformation will allow us to reuse the OPEN/CÆSAR environment and all the programs developed in CADP such as BISIMULATOR, ALDEBARAN, etc. We may structure the process of verifying C code in OPEN/CÆSAR in four phases:

2.1 Analyzing the C Code

The translation of the original system must start with an analysis phase. This phase is not trivial due to the complexity of the C language. In order to simplify this task, we first translate the C code into an intermediate XML-based language named PIXL.

Once the code is written in the mark up language, we perform an analysis focussing on two main issues. First, we need to extract every system call and control sentence for the creation of the *process graph*, described in the following section. Second, we need to analyze every variable in the program to determine whether it should be inserted in the state vector. In this case, the code has to be modified to reference the field of the state vector associated to the variable.

2.2 Creating the Process Graph

As mentioned above, our goal is to generate an implicit LTS from a C application containing external calls. In order to build the successor function needed to specify the LTS, we need a suitable representation of the different processes of the system to be analyzed. To this end, all these processes are converted into a graph before creating the implicit LTS.

We consider two types of labels. On the one hand, a label may represent a sequence of C statements (a block) that do not include any system call and, on the other, a label may represent a single system call.

Non determinism is an important aspect when representing system calls. Non determinism is implicit in communications, e.g. a broken connection, or failures in the calls to OS due to the impossibility of assigning a descriptor in a socket system call. Therefore, every system call with a nondeterministic behavior must be expressed by means of various transitions showing every possible behavior of the function.

Another point that must be taken into account is the translation of certain control sentences in the original C code. If the selection or iteration statements (if, case, while) contain API calls, we explicitly translate the structure of these sentences into the process graph. We express each selection sentence as the condition, the selection body, and the else branch. Similarly, each iteration sentence is defined as the condition and the body.

2.3 Generating the Implicit LTS

In the OPEN/CÆSAR environment, the generation of an LTS model from a C system involves the representation of states and labels of the system, and the implementation of the interface provided by CAESARGraph.h. This interface gives us the necessary primitives to manipulate states and labels that will be part of the final LTS.

Therefore, every global state, usually called state vector in model checking, contains the global data that may be accessed by all system processes as well as the local data corresponding to particular process instances. Global data include, for instance, channels in sockets or OS buffers. Local process variables are clearly local data. The global space also contains relevant data about the total number of processes running in the system. In addition, every process in execution keeps information about the actual state of the process, its pid, and the process type.

Besides the state representation, we must provide the label representatio. Labels represent actions to be carried out in order to evolve from one state to the following one. The label concept of CAESAR that have been inherited from LOTOS does not exists in C. In CAESAR a label is considered as a LOTOS gate and a number of experiments offered by the gate. However, in the socket case, we can find a direct relationship between the notions of gate and socket. Thus, the transition for a system call, e.g. a read call, generates a label similar to `read(5)`, where the number represents the socket identifier used for this communication.

Moreover, the generation of this interface requires two special primitives for constructing the transition relation of the LTS. One primitive is responsible for generating the initial state, and the other generates the successor states for any given state. The algorithm works in two phases. In the first one, for every system process, it explores all the transitions from its actual state. These transitions are obtained from the automaton associated to the process. The second phase of the algorithm executes each transition generated, appropriately updating the state vector variables and producing the corresponding LTS label. In order to execute the transition in the LTS, we need to execute the associated code in the process graph. Recall that this code is the result of the previous analysis described in Section 2.1. Thus, the successor state is generated automatically while executing the transition. It is worth noting that the transformed code works directly with the variables in the vector state.

The graph interface includes an initialization primitive `CAESAR_INIT_GRAPH`, which has to be called before using any operation with the graph. This method involves the allocation in memory of the process graphs (described in Section 2.2) of the different processes forming the system to be analyzed.

3 Conclusions and Future Work

Originally, OPEN/CÆSAR were designed as an open environment extending the functionality of CAESAR that is used for verifying a LOTOS specification. In this paper, we propose to use this framework for analyzing C programs that make use of well defined APIs. Most existing software model checkers are well suited to verifying distributed systems. Some of them, such as JPF or Bandera, analyze systems described in JAVA. Others are designed to analyze specific kinds of software. For instance, SLAM [1] verifies whether the behavior of a driver is secure wrt the uses of the API that it offers.

Actually, this proposal is its final stage of implementation, and the final version will be available in `http://www.lcc.uma.es/gisum/fmse/tools`. Future work could follow several lines. On the one hand, we could compare the results provided by our previous tool SOCKETMC, and by the current proposal. This comparison should take into account not only the numerical results, but also the easiness for obtaining the models. On the other, we also propose to extend our approach by considering different APIs or by implementing abstraction techniques.

References

1. Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, pages 1–20, 2004.
2. M. Camara, M.M. Gallardo, P. Merino, and D. Sanan. Model checking software with well-defined apis: The socket case. In *(FMICS05)*, pages 17–26. ACM SIGSOFT, 2005.
3. H. Garavel. OPEN/CAESAR: An open software architecture for verification, simulation, and testing. In *TACAS'98*, volume 1384, pages 68–84, 1998.
4. K. Havelund and T. Pressburger. Model checking java programs using java pathfinder, 1999.
5. Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
6. J. -C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: a protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 437–440, New Brunswick, NJ, USA, / 1996. Springer Verlag.