Extending CADP for Analyzing C Code [⋆]

M. Mar Gallardo[1], P. Merino[1] and D. Sanan[1]

LCC, Universidad de Malaga

Abstract. Many existing open source projects are written with the classic programming language C. Due to the size and complexity of such projects this applications require C-oriented methods and tools to increase their realibility. For instance, advanced reachability analysis techniques like *model checking*, that traditionally have been applied to software models, are now being considered as very promising methods to detect execution failures in final code. This paper focuses on extending the well known toolbox CADP in order to make it easier to analyze realistic concurrent C programs that make use of external functionality provided via well defined *application programming interfaces* (APIs). Our approach consists in constructing a tool to convert the C code into the usual formats expected by the set of tools integrating CADP (Construction and Analysis of Distributed Processes). The new module allows us to exploit all the functionalities of CADP to assist software reliability: model checking, equivalence checking, testing, distributed verification or performance evaluation.

1 Introduction

Currently, it is widely accepted that the different phases during the development of huge and complex software systems should be assisted by analysis tools, to ensure that the final product satisfies certain critical properties for the system under construction.

Research in the context of formal methods has provided specification and modelling languages, algorithms, tools and methodologies to automate diverse tasks that may help in the construction of good quality software. Nevertheless, due to time or memory constraints, most of these proposals apply to models (simpler descriptions/abstractions) of the real system to be executed. For instance, many enterprise information systems are partially or completely modeled to check some desired properties before the final system is implemented. An example of this are proposals [8] and [13], where authors model an Airport Terminal and a Suspendible Business Process, respectively.

Analysis during the software design phase is highly desirable, but it could also be very useful in the implementation phase since it could reveal new errors introduced in the (manual) code generation. Nowadays many academic projects, and even commercial ones, are working on adapting the model-oriented methods to most commonly used implementation languages. However, the number of tools that are currently able to deal with the reliability of final software is still very small, and the range of application is limited to very few tasks, such as verification using model checking with tools like SLAM [1] or JPF [12].

[⋆] Work partially supported by TIN2004-7943-C04-01 and TIN 2005-09405-C02-01

The goal of this paper is to extend the well known toolset CADP [6][1], adding new capabilities to assist in the development of reliable software, not only during the first stages of the lifecycle software, but also in the last stages, when encoding is being completed. The paper describes the successful development of our previous proposal in [4] towards using CADP as an environment for the analysis of C .

Toolset CADP offers several functionalities to manage specification languages (like LOTOS), such as compilers to CADP internal languages, equivalence checking, model checking, visualization of the execution graph, static analysis or performance evaluation. However, in order to extend all these functionalities and use them in a more complete software engineering process, we still need new compilers for standard programming languages.

CADP is an open platform which allows users to integrate new specification, verification or analysis techniques. It provides libraries that may be used to extend the toolbox at different levels. Thus, our tool C.Open[2] permits reusing the different modules offered by CADP to analyze C programs. To do this, C.Open translates the C code into an implicit labelled transition system (LTS), which is the CADP internal format. Furthermore, C.Open is specially oriented to analyzing C programs with calls to well defined APIs. As explained in [2], the analysis of software with calls to external application programming interfaces (APIs) makes it necessary to construct models of all the functions provided by the API. In the paper, we provide a scheme to model APIs which is compatible with CADP architecture.

Through several examples, the paper shows how we can use different functionalities of the CADP environment (such as explicit graph generation, reduction or simulation) to analyze C programs. Although the proposal is applicable to generic APIs, the examples make use of a specific API that provides functions to correctly share memory regions between C processes. During translation, calls to this API are substituted by models of its behavior written in C. Our compiler generates the necessary data structures to exploit all the features in CADP. Moreover this is the base to obtain C-oriented tools like model checkers or tools for testing equivalence.

The paper is organized as follows. Section 2 gives an introduction on OPEN/CAESAR [5]. Section 3 illustrates the input language of the tool, and the kind of code that C.Open can manage. Section 4 shows the use of C.Open with an example consisting of two C concurrent programs that communicate via shared memory. Finally, Section 5 gives the conclusions and future work.

2 CADP Overview

CADP can be considered as a traditional toolbox for the analysis of communication protocols. Through a modular architecture, CADP includes compilers to translate several input formalisms into a generic format (an LTS) which is used by applications as an internal representation of the input language. Figure 1 shows the different formalisms accepted by CADP inside boxes, and dashed lines represent the different compilers.

[1] CADP web site: “http://www.inrialpes.fr/vasy/cadp.html”
[2] C.Open web page: “http://www.lcc.uma.es/gisum/tools/smc”

As shown in the figure, C.Open adds language C to the CADP architecture as alternative input. The modular structure of CADP makes it possible to reuse the whole set of applications present in the environment.

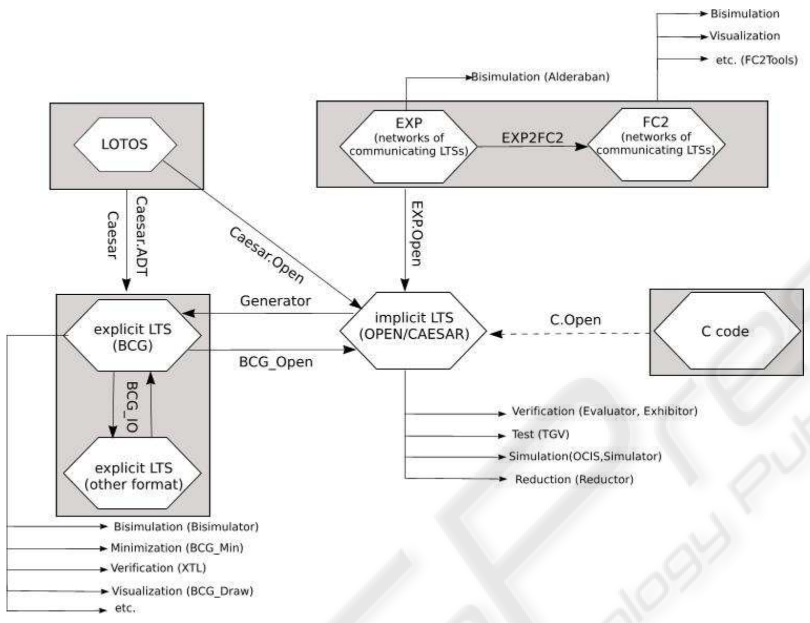


Fig. 1. Schema of the extended CADP architecture including C.Open.

2.1 Labelled Transition Systems: The Internal Format

As commented above, the different tools in CADP accept models of systems described as LTSs. An LTS is a tuple (S, L, T, q_0) where S is the set of system states, and $q_0 \in S$ is the initial state. Set $T \subset S \times L \times S$ defines the transition relation, L being a set of labels used to identify transitions. As usual, transition $(q, l, q') \in T$ is written as $q \xrightarrow{l} q'$, and it represents system evolution from state q to q' by executing sentence l.

Labels may represent instructions dealing with global data structures used to communicate or synchronize processes or, on the contrary, they may refer to internal actions in a given process. We use the special label ι to generically denote all these local actions. Thus, in C.Open, labels ι represent transitions involving only C statements that do not contain any external call.

The CADP input language may be transformed into an *implicit* or an *explicit* LTS. The implicit representation of an LTS consists of C representations of the states and labels along with the necessary primitives to handle them. It also provides primitives to compute the initial state and the successors of any given state. The explicit representation records the LTS graph, storing the whole set of transitions T. In an LTS of several hundreds of thousand of states, this representation may be too big.

Since real software written in C may produce very large LTSs, we have used the implicit representation when extending CADP with C oriented tools. The implicit LTS is given by two primitives to handle the corresponding transition relation: a function to obtain the initial state of the system, and a function to generate all the successor states of any given state. In addition, the implicit LTS provides primitives to print, compare and hash generation of states and labels.

2.2 CADP Modules

CADP includes a wide set of tools providing different functionalities. For example, it contains a module to analyze whether two LTSs are bisimilar. It also provides several model checkers for various temporal logics and for μ-calculus. It implements several verification algorithms including exhaustive verification, on-the-fly verification, symbolic verification using Binary Decision Diagrams, and compositional verification based on refinement. CADP has been recently extended with other programming language oriented tools like ANNOTATOR [3].

Some of the tools in CADP are particularity interesting for the software engineering community. For instance, EVALUATOR (model checker for mu-calculus formulas), TGV (generator of conformance test suites), BISIMULATOR (checker of equivalence relations), REDUCTOR (LTS on-the-fly reduction with respect to a relation), EXHIBITOR (search patterns of execution sequences), OCIS and SIMULATOR (graphical and command-line simulators, respectively).

2.3 Extending CADP

CADP is not only a set of tools, but also a tool development framework. OPEN/CÆSAR is an interface for the creation of new modules in the CADP toolkit. OPEN/CÆSAR separates the functionalities of each application in three different modules: the graph, the storage and the exploration modules. The exploration module performs the basic functionality of the application and the operations needed to handle the storage and graph modules. The storage module is constituted by a set of libraries, included in OPEN/CÆSAR, representing several structures to store the labels and states of the LTS. Finally, the graph module provides the exploration module with the necessary operations to handle the implicit LTS, that is, to handle states, labels and to generate the successor states.

CADP only provides compilation for some specification languages, such as LOTOS or binary code graphs (BCG), through the tools CAESAR.OPEN and BCG_OPEN; however, as commented above, it does not support programming languages. C.Open, based on [4], extends CADP, making it possible to use the whole environment with C programs.

3 C.Open Input Language

Most proposals to formally analyze C code only consider closed C programs, that is, programs where the implementation of all functions is available to be executed by the analysis tool. In this context, tool C.Open provides a new functionality, since it can manage C programs that make calls to an external API. Furthermore, the programs can actually use the external API to form a concurrent system. Given this concurrent system, C.Open may construct its state spaces on-the-fly.

In order to deal with external APIs, we need to construct *models* of the external calls. These models are, in fact, abstractions of the real behavior of the external functions. They only provide the minimum functionality required to carry out the analysis. For instance, if an external function deals with intermediate communication buffers, we probably do not need to implement buffers with their real size, it might be sufficient to use some type of reduced buffers. In fact, due to the state space problem, reducing the complexity of real data structures is essential to obtain effective analysis tools, like model checkers. The models of external calls are C functions which are executed by the graph module when it is generating the successors of a given state if any of them correspond to one external function.

The decision about how to model external functions strongly depends on the properties to be analyzed on the system. In any case, the transitions of the resulting LTS may be labelled with ι representing an atomic sequence of internal C sentences (including no external call) or, on the contrary, they may be labelled with a call to the model of an external function. In summary, we can deal with any kind of C code provided models of all external calls to the corresponding API are given. The next section explains how to obtain models of external functions.

3.1 The External API

Table 1. Shared Memory API functions.

func.	return	arg 1	arg 2	arg 3
screate	reg.id(int)	reg name(char *)	sizeof reg.(int)	value(void *)
sread	value(void *)	reg.id(int)		
swrite	code(int)	reg.id(int)	value(void *)	sizeof value(int)
sclose	code(int)	reg.id(int)		

For external calls to the language, C.Open needs a model of these functions written in C and a translation rules file for translating the external API functions into the corresponding modelled function. Table 1 shows, as an example, the API `Shared Memory`, also used in Section 4, that provides four basic functions to deal with a shared resource, that is, `create`, `read`, `write` and `close`. The shared memory is composed of several regions, each one with a unique name and size. When `screate` is called with a given name, size and initial value, a new region is created, provided that no region has been previously created with the same name, size and initial value. Otherwise, if there was a region with the same name, the function call returns the region identifier previously assigned. The other operations, `sread`, `swrite` and `sclose`, are used to read from, write to, or close the region specified by the corresponding argument. In particular, the `sclose` operation decreases the number of references to that region, deallocating the reserved memory if there are no references left. Any attempt to access a non-existent region returns an error code.

C.Open makes use of the so-called *translation rules* to properly transform each external function. These rules are given in an XML file where, for each function call, the arguments that must be preserved or that must be added in the modelled function are specified. For example, Figure 2 shows the translation rules for the function `sread`. It indicates that `sread` is translated into function `read_shared_memory`, which has two arguments: the first one refers to the first argument of `sread`, and the other is the value returned by the function. It is possible that an argument may have a different representation in the label of the LTS, being represented by the name of the variable instead of the real variable value, like the first argument of the `sread`.

```
<function name="sread" sname="read_shared_memory" type="1"> <arg
typeArg="1" argref="0" type="int" labeltype="char"
     varname="yes" labelsize="20" labelname="desc"/>
<arg typeArg="0" type="void *" labeltype="int" returned="true"/>
</function>
```

Fig. 2. sread translation rules.

4 Example

In order to highlight the benefits of the translation from C to LTS, we show how the different CADP tools can be used to analyze C programs with calls to an external API. In particular, the API example in section 3.1 will be used together with a model of this API. We will show how C.Open works implementing the Peterson's mutual exclusion algorithm [11] and using several CADP tools as generator, simulator or evaluator to prove the correctness of the programs. This algorithm can be used, for example, as a mechanism to ensure data consistency in multi-user database systems, which are present in many enterprise information systems.

4.1 The Sample Program

The system to be analyzed is composed of two programs, p0_peterson.c (figure 3) and p1_peterson.c, that use the Peterson mutual exclusion algorithm for access to a common critical section. Both programs are symmetrical, they only differ in the pid, and in the control flag variables that guard the critical section. The program begins with the creation of the shared variables that control the critical section. Before going into the critical section, both processes make an active wait for the critical section. Before closing the shared variables, each process updates the flag shared variable to ensure that the other process can not exit from the active wait.

4.2 Generating the Explicit Graph using Generator

C.Open generates an executable application (e.g., generator.exe) by performing the required sequence of tool invocations: translation by C2Xml of C programs into PIXL [9]

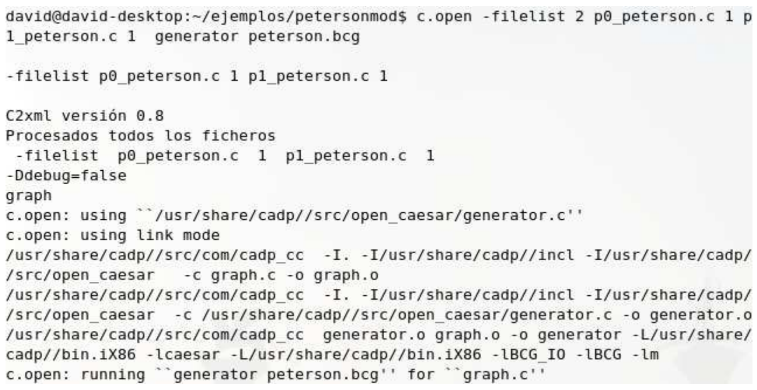
```
int main (int argc, char **argv) {
 unsigned int flag0_des, flag1_des, turn_des;
 int flag0_value, flag1_value, turn_value;
 int flag0_res, flag1_res, turn_res;
 int pid, initial_value;

 /* Local process identification */
 initial_value = 0;
 pid = initial_value;
 /* Initialization of shared variables */
 flag0_des = screate ("flag0", /* descriptor name for flag0 */
              sizeof (flag0_value),    /* value size of flag0 */
              &initial_value /* initial value for flag0 */ );
 flag1_des = screate ("flag1", /* descriptor name for flag1 */
              sizeof (flag1_value),    /* value size of flag1 */
              &initial_value /* initial value for flag1 */ );
 turn_des = screate ("turn",   /* descriptor name for turn */
             sizeof (turn_value),  /* value size of turn */
             &initial_value /* initial value for turn */ );
 /* Behavior of process 0 */
 flag0_value = 1;
 flag0_res = swrite (flag0_des,    /* descriptor for flag0 */
             &flag0_value, /* pointer to flag0 value */
             sizeof (flag0_value)  /* value size of flag0 */);
 turn_value = 1;
 turn_res = swrite (turn_des,  /* descriptor for turn */
            &turn_value,   /* pointer to turn value */
            sizeof (turn_value)    /* value size of turn */);
 /* Busy waiting for remote process */
 pid = (pid + 1) % 2;
 while ((*(int *) sread (flag1_des /* descriptor for flag1 */ ) == 1) &&
    (*(int *) sread (turn_des /* descriptor for turn */ ) == 1))
   {
     printf ("Waiting for process %d\n", pid);
   }
 /* Critical section */
 pid = (pid + 1) % 2;
 printf ("Process %d is in critical section\n", pid);
 /* End of critical section */
 flag0_value = 0;
 flag0_res = swrite (flag0_des,    /* descriptor for flag0 */
             &flag0_value, /* pointer to flag0 value */
             sizeof (flag0_value)  /* value size of flag0 */);
 /* Close shared memory */
 flag0_res = sclose (flag0_des /* descriptor for flag0 */ );
 flag1_res = sclose (flag1_des /* descriptor for flag1 */ );
 turn_res = sclose (turn_des /* descriptor for turn */ );
```

Fig. 3. Peterson’s mutual exclusion algorithm using shared memory.

compliant XML models; slicing of the models with respect to the system API and construction of the OPEN/CÆSAR graph module describing the implicit LTS by C2Lts; and finally, call to the C compiler.

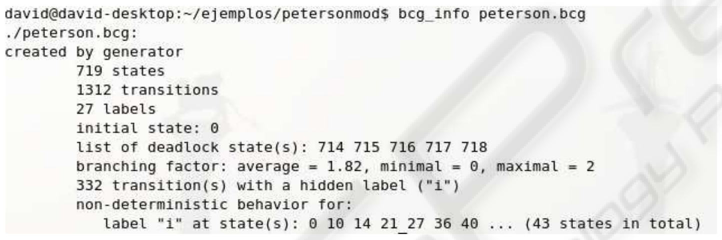
In figure 4 C.Open generates and invokes the executable for generator. The command line C.Open takes as arguments the input for C.Open and the exploration module, GENERATOR in this example, with the corresponding parameters (i.e. the file where GENERATOR will save the bcg generated).

Figure 5 shows the caption of the info for the bcg created by GENERATOR. It has 719 states, but CADP includes several tools to reduce the graph through bisimulation, being easier to manage and represent that way. Among these applications, REDUCTOR performs an exhaustive analysis and generates the LTS corresponding to an input bcg. The resulting LTS is reduced on-the-fly respect to several relations (strong equivalence, tau-divergence, tau-compresion, tau-confluence, tau*.a equivalence, safety equivalence, trace equivalence, or weak trace equivalence). So, if we apply REDUCTOR to the bcg obtained after applying GENERATOR with a total reduction, we get a smaller equivalent LTS, figure 6, with only 157 different states and 288 transitions.



```
david@david-desktop:~/ejemplos/petersonmod$ c.open -filelist 2 p0_peterson.c 1 p
1_peterson.c 1  generator peterson.bcg

-filelist p0_peterson.c 1 p1_peterson.c 1

C2xml versión 0.8
Procesados todos los ficheros
 -filelist  p0_peterson.c  1  p1_peterson.c  1
-Ddebug=false
graph
c.open: using ``/usr/share/cadp//src/open_caesar/generator.c''
c.open: using link mode
/usr/share/cadp//src/com/cadp_cc  -I. -I/usr/share/cadp//incl -I/usr/share/cadp/
/src/open_caesar   -c graph.c -o graph.o
/usr/share/cadp//src/com/cadp_cc  -I. -I/usr/share/cadp//incl -I/usr/share/cadp/
/src/open_caesar  -c /usr/share/cadp//src/open_caesar/generator.c -o generator.o
/usr/share/cadp//src/com/cadp_cc  generator.o graph.o -o generator -L/usr/share/
cadp//bin.iX86 -lcaesar -L/usr/share/cadp//bin.iX86 -lBCG_IO -lBCG -lm
c.open: running ``generator peterson.bcg'' for ``graph.c''
```

Fig. 4. Call to C.Open to generate an explicit LTS with generator.



```
david@david-desktop:~/ejemplos/petersonmod$ bcg_info peterson.bcg
./peterson.bcg:
created by generator
        719 states
        1312 transitions
        27 labels
        initial state: 0
        list of deadlock state(s): 714 715 716 717 718
        branching factor: average = 1.82, minimal = 0, maximal = 2
        332 transition(s) with a hidden label ("i")
        non-deterministic behavior for:
           label "i" at state(s): 0 10 14 21_27 36 40 ... (43 states in total)
```

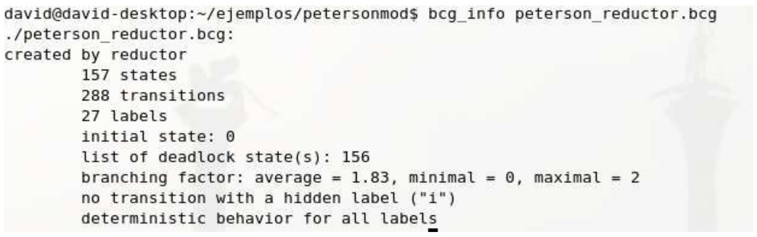
Fig. 5. Information of the explicit LTS generated by generator.

4.3 Simulating with Simulator and Executor

It is possible to use SIMULATOR and EXECUTOR for simulating C programs. With SIMULATOR, we can perform a guided execution of the analyzed programs. From one state, it is possible to execute one transition, representing an external call or a set of C sentences without any of the modeled calls, backtrack to a previous state, view the actual system state or the execution trace. Figure 7 shows a simulation example with XSIMULATOR. EXECUTOR, on the other hand, performs a random execution, showing as a result the final execution path.

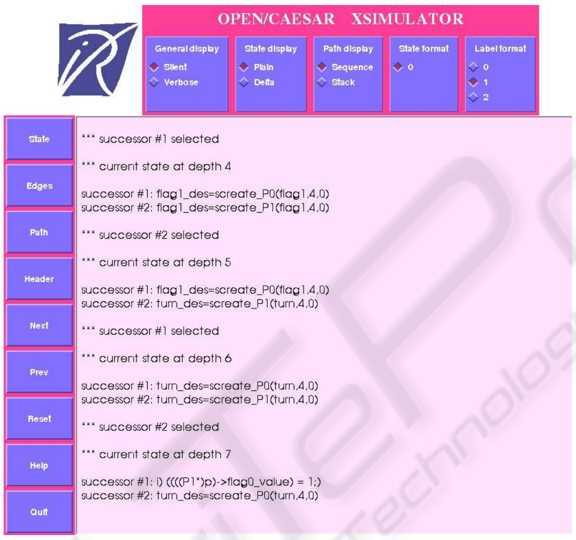
5 Conclusions and Future Work

C.Open permits the use of the environment provided by CADP for the automatic analysis of C code. Our approach to extend CADP directly allows us to perform different kinds of analysis of the C code, like model checking, simulation, bisimulation or static analysis. New C-oriented functionalities can now be implemented for CADP. Other proposals for analyzing C code focus only on one functionality, like model checking (CMC [10] or SLAM [1]) or debugging (gdb [7]).



```
david@david-desktop:~/ejemplos/petersonmod$ bcg_info peterson_reductor.bcg
./peterson_reductor.bcg:
created by reductor
        157 states
        288 transitions
        27 labels
        initial state: 0
        list of deadlock state(s): 156
        branching factor: average = 1.83, minimal = 0, maximal = 2
        no transition with a hidden label ("i")
        deterministic behavior for all labels
```

Fig. 6. Information of the explicit LTS after being reducted with reductor.



General display: [✓ Silent] Verbose
State display: [✓ Plain] Delta
Path display: [✓ Sequence] Stack
State format: [✓ 0]
Label format: 0 [✓ 1] 2

```
*** successor #1 selected

*** current state at depth 4

successor #1: flag1_des=screate_P0(flag1,4,0)
successor #2: flag1_des=screate_P1(flag1,4,0)

*** successor #2 selected

*** current state at depth 5

successor #1: flag1_des=screate_P0(flag1,4,0)
successor #2: turn_des=screate_P1(turn,4,0)

*** successor #1 selected

*** current state at depth 6

successor #1: turn_des=screate_P0(turn,4,0)
successor #2: turn_des=screate_P1(turn,4,0)

*** successor #2 selected

*** current state at depth 7

successor #1: i) ((((P1*)p)->flag0_value) = 1;)
successor #2: turn_des=screate_P0(turn,4,0)
```

Fig. 7. Simulating the application with xsimulator.

As future work, new lines for code analysis can be added, for example, optimization techniques like partial order reduction to reduce the number of states generated, or the research to deal with dynamic memory. Another point of interest is the automatic generation of API models.

More information and upcoming extensions of our tool will be available at “http://www.lcc.uma.es/gisum/tools/smc”.

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 M. Massink. T. Margaria, editor, *Proc. of the Tenth In-*

ternational Workshop on Formal Methods for Industrial Critical Systems (FMICS05), pages 17–26. ACM SIGSOFT, 2005.

3. M.M Gallardo, C. Joubert, and P. Merino. Implementing influence analysis using parameterised boolean equation systems. In Nicolas Halbwachs and Lenore Zuck, editors, *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation ISOLA'06 (Paphos, Cyprus)*, volume 3440 of *Lecture Notes in Computer Science*, pages 581–585. IEEE Computer Society Press, November 2006.
4. M.M. Gallardo, P. Merino, and D. Sanan. Towards model checking c code with open/caesar. In *Proc. of MSVVEIS'06*, pages 198–201, 2006.
5. H. Garavel. OPEN/CAESAR: An open software architecture for verification, simulation, and testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98*, volume 1384, pages 68–84, 1998.
6. Garavel, H., Lang, F., and Mateescu, R. An overview of cadp 2001. In *EASST Newsletter*, number 4.
7. http://sourceware.org/gdb/. *GDB, the GNU project debbuger*.
8. I. Manataki and K. Zografos. A system dynamics approach for airport terminal performance evaluation. In *Proc. of MSVVEIS'06*, pages 206–209, 2006.
9. Gallardo M.M, Martnez J., Merino P., Nuez P., and Pimentel E. Pixl: Applying xml standards to support the integration of analysis tools for protocols. *Science of Computer Programming*, 65:57–69, March 2007.
10. Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.
11. Michel Raynal. *Algorithmique du parallelisme : le probleme de l'exclusion mutuelle*. 1984.
12. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *IEEE Computer Society*, pages 3–12, Grenoble,France, sep 2000.
13. Yeung W., Leung K., Wang J., and Dong W. Modelling and model checking suspendible business processes via statechart diagrams and csp. *Science of Computer Programming*, 65:14–29, March 2007.