C.OPEN and ANNOTATOR: Tools for On-the-Fly Model Checking C Programs

María del Mar Gallardo[1], Christophe Joubert[2], Pedro Merino[1], and David Sanán[1]

[1] University of Málaga, Campus de Teatinos s/n, 29071, Málaga, Spain
{gallardo,pedro,sanan}@lcc.uma.es
[2] Technical University of Valencia, Camino de Vera s/n, 46022, Valencia, Spain
joubert@disc.upv.es

Abstract. This paper describes a set of verification components that open the way to perform on-the-fly software model checking with the CADP toolbox, originally designed for verifying the functional correctness of LOTOS specifications. Two new tools (named C.OPEN and ANNOTATOR) have been added to the toolbox. The approach taken fits well within the existing architecture of CADP which doesn't need to be altered to enable C program verification.

1 Introduction

The software model checking problem consists in verifying that a program, *i.e.* an infinite state system described in a high-level language, does not contain errors, such as improper memory access, misuse of system interfaces, or violation of (temporal logic) properties. The verification process is automatic, and the wrong conception of the program is eventually illustrated by means of potential offending behaviors of the system (*e.g.*, counter examples). Traditionally, programs are first analysed to statically remove parts that do not affect the property of interest, using light-weight pre-processing technique such as *program slicing*. The reduced model is then abstracted using *predicate abstraction* and *localization* techniques. Finally, the resulting finite state system is processed by SAT-based, BDD-based or explicit state model checkers.

Existing software model checkers, like SLAM [1] and BLAST [2], are either domain specific (*e.g.*, verification of drivers), language dependent, or based on dedicated algorithms and tools. This paper presents an analysis engine that finds application programming interface (API) usage errors in C programs, similarly to BLAST and SLAM, but rather focusing on a general purpose model checking framework. We describe a set of components, namely C.OPEN and ANNOTATOR, that enable the explicit state verification of C programs by means of the last stable CADP 2006 "Edinburgh" release. The CADP toolbox[3] [3] is a complex software suite integrating numerous verification tools. CADP supports the process algebra LOTOS for specification, and offers various tools for simulation and formal verification, including equivalence checkers (bisimulations) and model checkers (temporal logics and modal μ-calculus). The toolbox is designed as an

[3] CADP web site: "http://www.inrialpes.fr/vasy/cadp".

open platform for the integration of other specification, verification and analysis techniques. This is realized by means of APIs which on different levels provide means to extend or exploit the functionalities of the toolbox. These APIs have been used by others to link CADP to other specification languages as well as other verification/testing tools. Here we describe how these APIs have been used by C.OPEN to support C program transformation and abstraction based on XML intermediate representation, and by ANNOTATOR to support on-the-fly data flow analysis and program slicing, namely *influence analysis*, of implicit control flow graphs using boolean equation systems (BESs). Our efforts have been driven by the intention to avoid changes to the existing components as much as possible, while providing a sound and efficient framework for C program model checking.

Originality. Our approach differs from previous works, like BLAST and SLAM, in several ways:

1. in our connection to the CADP toolbox of the first model generator (C.OPEN) that automatically extracts implicit LTS models from programs written in C programming language, and of the first on-the-fly static analyzer (ANNOTATOR),
2. in our emphasis on the verification of distributed protocols (*e.g.*, the Peterson's mutual exclusion (PME) protocol between two processes), using well-specified APIs, described as multiple (or multi-instantiated) concurrent independent C programs, rather than on sequential (SLAM) or multi-threaded programs (ongoing work of BLAST),
3. in our use of open-modular architectures (OPEN/CÆSAR) and technologies (XML, BES, LTS) to represent the state-space and verification problem efficiently and to facilitate the connection to other programming languages, like Promela, and
4. in the way we concentrated this research work on the compiler side and used well-established verification tools of the CADP toolbox as back-end.

2 Software Architecture

The toolset encompasses two sorts of tools (see Figure 1) to verify C programs generated via CADP. (i) The C.OPEN tool provides different means to distill an implicit labeled transition system (LTS) from a C program. (ii) The static analyser ANNOTATOR enables on-the-fly data flow and influence analysis of implicit LTSs describing abstract control flow graphs (CFGs).

Distilling implicit LTSs from a C program. C.OPEN [4] is an addon component for CADP to support C program input to the OPEN/CÆSAR environment [5], though we state that the XML API, called PIXL [6], on which the tool is based, is general enough to attach the C program abstraction process to other verification toolboxes, such as SPIN, via Promela specifications instead of implicit LTSs [7]. The idea that OPEN/CÆSAR environment can be connected to a C compiler and that existing CADP tools can thereby be extended to this new class of

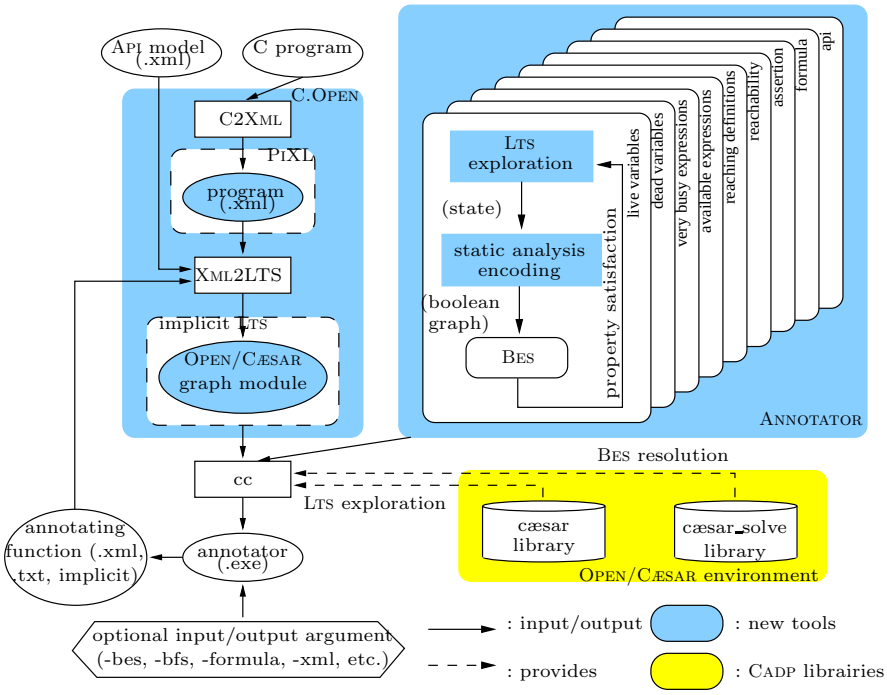

Fig. 1. C.OPEN and ANNOTATOR tools

specifications is an important step towards re-using well-established verification toolboxes. C.OPEN (400 lines of Shell script) takes as inputs a system described by a set of C programs, an operating system API's model represented in XML, and an OPEN/CÆSAR application (*e.g.*, ANNOTATOR). As an output, it generates an executable application (*e.g.*, annotator.exe) by performing the required sequence of tool invocations: 1) a tool, called C2XML (2 000 lines of JAVA), is used with JAVACC and a C grammar (1 000 lines of JAVA), to translate C programs into PIXL compliant XML models; 2) another tool, called XML2LTS (4 500 lines of JAVA), then slices the program models with respect to system APIs to be preserved and it constructs the OPEN/CÆSAR graph module describing the implicit program LTS; 3) finally, the C compiler *cc* is called.

C.OPEN allows to construct abstracted state spaces on-the-fly, and only to the required precision (*w.r.t.* a specific API). It currently offers the possibility to generate either CFG or explicit state space of a program as an implicit LTS.

Analysing implicit CFGs. ANNOTATOR implements standard data flow analysis algorithms on a CFG, by using boolean equation systems (BESS) [8]. It also computes various influence analyses [3], generally used for compacting the program state representation, by detecting the relevant program variables in each control point, for a property of interest.

Our static analyser takes as inputs a static analysis to carry out and an LTS describing the CFG of a program, in which instructions are abstracted to the strict necessary information (*i.e.*, modified and defined variables, used expres-

sions, and instruction type). This LTS is represented implicitly by its successor function as an OPEN/CÆSAR program provided by compliant compilers, such as C.OPEN, but existing CADP compilers, such as CÆSAR, could be directly extended to provide such CFGs [10].

ANNOTATOR (6 000 lines of C code) consists of several modules, each one containing the BES translation for a particular static analysis (live variables, very busy expressions, available expressions, reaching definitions, reachability, assertion control, formula and API preservation influence analyses). BESs are represented implicitly by their successor function, in the same way as LTSs in OPEN/CÆSAR. They are handled internally by the CÆSAR_SOLVE [11] library, which offers several on-the-fly resolution algorithms, based on different search strategies (*e.g.*, breadth-first). Dependent on the option selected by the user, the analysis result is written to an XML or textual file. These formats allow post-processing of computed analyses, by directly conveying the result as input to compilers reading these formats, such as C.OPEN, allowing further compilation optimizations.

Availability. The proposed tools are publicly available through the following web pages http://www.lcc.uma.es/gisum/tools/smc. C.OPEN and ANNOTATOR, being part of the database of research tools developed using CADP, are also referenced by the CADP web site. Both new tools are rather small, robust and mature (in operation for about a year) and detailed manual pages are provided, as well as more than 25 program examples and step-by-step small case studies.

Applicability. Concerning applicability, C.OPEN compiles concurrent C programs into the OPEN/CÆSAR intermediate format (*i.e.*, implicit labeled transition system (LTS)), to which efficient CADP model checkers, such as EVALUATOR (evaluation of regular alternation-free μ-calculus formulas) and BISIMULATOR (equivalence checking), are connected. Hence, CTL, ACTL, PDL, PDL-Δ and regular alternation-free μ-calculus properties can be verified on our C input programs. In the PME demonstration, we successfully checked respectively one safety, liveness and fairness property on the C implementation of the protocol and we also reduced the explicit-state space size by 20% using API influence analysis results computed by the ANNOTATOR tool. Furthermore, all analyses that are available in the CADP toolbox can be directly used on our C input programs.

Concerning comparison with well-established software model checkers, like BLAST or SLAM, since they cannot deal with distributed protocols with well-defined APIs, a direct comparison of our tools with them would be irrelevant.

Scalability. ANNOTATOR has been successfully experimented on very large CFGs, extracted from the VLTS benchmark[4], with size up to 10^6 program counters and instructions. Moreover, C.OPEN and ANNOTATOR allow several levels of abstraction of the program instructions present in the LTS model, giving the

[4] VLTS web site: http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html

possibility to verify further properties or to achieve further reductions on the program model.

3 Conclusion and Future Work

The development of an on-the-fly software model checker "from scratch" is a complex and costly task. The open modular architecture adopted for C.Open and Annotator aims at making this process easier, by using the Xml intermediate representation, the well-established verification framework of Bess, together with the generic libraries for Lts exploration and Bes resolution provided by Cadp. For instance, this tool architecture reduces the effort of implementing a new static analysis to its strict minimum: encoding the mathematical definition of the analysis as a Bes, and interpreting the result. We plan to continue our work by extending Annotator with other static analyses (e.g., reset variables analysis [10]) and by interconnecting the two new Cadp components with tools extending Spin, such as SocketMC [7] and αSpin [12].

References

1. Ball, T., Rajamani, S.K.: The slam toolkit. In Proc. of CAV'01 (Paris, France). LNCS Vol. 2102, pp. 260–264. Springer Verlag.
2. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In Proc. of CAV'06 (Seattle, WA, USA). LNCS Vol. 4144, pp. 532–546. Springer Verlag.
3. Garavel, H., Lang, F., Mateescu, R.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In Proc. of CAV'07 (Berlin, Germany). To appear. LNCS. Springer Verlag.
4. Gallardo, M., Merino, P., Sanán, D.: Towards model checking c code with open/cæsar. In Proc. of MSVVEIS'06 (Paphos, Cyprus). pp. 198–201. Insticc.
5. Garavel, H.: Open/cæsar: An open software architecture for verification, simulation, and testing. In Proc. of TACAS'98 (Lisbon, Portugal). LNCS Vol. 1384, pp. 68–84. Springer Verlag.
6. Gallardo, M., Martínez, J., Merino, P., Nuñez, P., Pimentel, E.: Pixl: Applying xml standards to support the integration of analysis tools for protocols. Science of Computer Programming. In Press. 2006
7. Camara, P., Gallardo, M., Merino, P., Sanán, D.: Model checking software with well-defined apis: the socket case. In Proc. of FMICS'05 (Lisbon, Portugal). pp. 17–26. ACM-SIGSOFT.
8. Gallardo, M., Joubert, C., Merino, P.: Implementing influence analysis using parameterised boolean equation systems. In Proc. of ISOLA'06 (Paphos, Cyprus). IEEE Computer Society Press.
9. Cámara, P., Gallardo, M., Merino, P.: Abstract matching for software model checking. In Proc. of SPIN'06 (Vienna, Austria). LNCS Vol. 3925, pp. 182–200. Springer.
10. Garavel, H., Serwe, W.: State space reduction for process algebra specifications. Theoretical Computer Science **351**(2):131–145. 2006.
11. R. Mateescu. Caesar_solve: A generic library for on-the-fly resolution of alternation-free boolean equation systems. *Springer Int. J. on Soft. Tools for Tech. Trans. (STTT)*, 8(1):37–56, 2006.
12. Gallardo, M., Martinez, J., Merino, P., Pimentel, E.: αspin: A tool for abstraction in model checking. Software Tools for Technology Transfer **5**(2-3):165–184. 2004.

C.OPEN and ANNOTATOR: Tools for On-the-Fly Model Checking C Programs

María del Mar Gallardo, Christophe Joubert, Pedro Merino and David Sanán

University of Málaga, Campus de Teatinos s/n,
29071, Málaga, Spain
{gallardo,pedro,sanan}@lcc.uma.es
Technical University of Valencia, Camino de Vera s/n,
46022, Valencia, Spain
joubert@dsic.upv.es

Appendix of tool paper submission for SPIN 2007

This appendix contains information about a potential demonstration of the software model checking toolset at SPIN 2007. Among the case studies we have pursued so far with this toolset, we have selected a small example that illustrates the entire tool chain and methodology, and touches most of the tool components discussed in the paper. It is based on an implementation in C code of the Peterson's mutual exclusion (PME) algorithm, taken from [1], for mutual exclusion between two processes.

In the following we present details of a potential series of experiments and results, of methods and implications for the Peterson's mutual exclusion example to be demonstrated at the conference.

A Case study: The Peterson's mutual exclusion

PME is a concurrent programming algorithm for mutual exclusion that allows just two processes to share a single-use resource without conflict, using only shared memory for communication. It was formulated by Gary Peterson in 1981 at the University of Rochester.

To prevent a same piece of data from being in an inconsistent and unpredictable state, *critical sections* of code accessing shared data must therefore be protected, so that other processes which read from or write to the data are excluded from running.

The PME algorithm (see Figure 2) uses two global variables, *flag* and *turn*. A *flag* value of 1 indicates that the process wants to enter the critical section. The variable *turn* holds the identification of the process whose turn it is. Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting *turn* to 0.

The algorithm satisfies the three essential criteria of mutual exclusion:

- *Mutual exclusion.* P0 and P1 can never be in the critical section at the same time: If P0 is in its critical section, then either *flag*[1] is false or *turn* is 0. In both cases, P1 cannot be in its critical section.
- *Progress requirement.* If process P0 does not want to enter its critical section, P1 can enter it without waiting. There is not strict alternating between P0 and P1.
- *Bounded waiting.* A process will not wait longer than one turn for entrance to the critical section: After giving priority to the other process, this process will run to completion and set its *flag* to 0, thereby allowing the other process to enter the critical section.

Although global variables *flag* and *turn* are shared resources used to communicate between the two concurrent processes, only variable *turn* could get corrupted being modified from both processes. Since it always keeps a consistent

```
flag[0] = 0
flag[1] = 0
turn = 0

p0:  flag[0] = 1                          p1:  flag[1] = 1
     turn = 1                                  turn = 0
     while( flag[1] && turn == 1 );            while( flag[0] && turn == 0 );
         // do nothing                             // do nothing
     // critical section                       // critical section
     ...                                       ...
     // end of critical section                // end of critical section
     flag[0] = 0                               flag[1] = 0
```

Fig. 2. Peterson's mutual exclusion algorithm

value through the algorithm execution, it is not necessary to use specific mutual exclusion mechanism to manipulate these global variables. The only critical section is then the one indicated in the algorithm.

In order to confirm that the PME algorithm satisfies the three essential criteria of mutual exclusion, we intend to study the functional correctness of the algorithm by means of an abstract model. Here we construct both a model of the shared memory and the explicit state space of the implemented algorithm using the latest stable release of CADP 2006 "Edinburgh" for labeled transition system exploration.

For this, we use the following C implementation of PME algorithm, where we only present P0 implementation (*p0_peterson.c*), P1 being symmetric. In short, the implementation consists in creating and initializing global variables, appropriately setting flags, and testing condition to enter the critical section. When exiting the critical section, the process sets its flag to 0, allowing the waiting process to enter the critical section. Furthermore, each global variable that was used by the exiting process, gets its number of processes using it decreased by one.

```
/*****************************************************************************
 *       THE PETERSON'S MUTUAL EXCLUSION ALGORITHM      ( PROCESS 0 )
 *---------------------------------------------------------------------------
 *   Module             :       p0_peterson.c
 *   Authors            :       Christophe JOUBERT and David SANAN
 *   Version            :       1.2
 *   Date               :       07/01/27 17:07:42
 *****************************************************************************/

/* Note: a real application would perform error checking */
#include <stdio.h>

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 of 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 */ );
  /*************************************************************************/
}
```

This program describes the functional behavior of process P0 as well as the various system calls to the shared memory model. Our three global variables *flag0*, *flag1*, and *turn* are represented as shared memory locations. The XML file of Figure 3, called *externalfunctions.xml*, describes the API of the shared memory model. In short, *screate* is used to allocate and initialize memory space for each global variable, or if already existing, updating the number of active processes on the shared resource. It returns a descriptor identifying the memory location. *sread* and *swrite* respectively gets or modifies the memory content pointed by a descriptor. Finally, *sclose* decreases the number of active processes on the shared memory location pointed by a descriptor, and if the resource becomes unused, it frees the corresponding memory space. *sread*, *swrite*, and *sclose* return a successful or error code upon completion.

The concrete behavior of the shared memory model is implemented in C code in files *sharedmemory[.c,.h]* accessible from the toolset web pages.

We use the new tool C.OPEN, to compile the C implementation of the PME algorithm together with the C and XML description of the shared memory model. Together with the GENERATOR application of CADP, it generates the PME explicit-state space, stored in BCG format. This later has 719 states and 1 312 transitions.

> *The tool demonstration will show how the state space is generated using the* EUCALYPTUS *interface of* CADP *exhibiting on-the-fly monitoring of the generation (see Figure 4).*

While the CADP toolset has originally been developed to support design and verification of functional properties of LOTOS specifications, the emphasis of this study is to apply the CADP verification framework to C programs, in particularly, to an implementation of the PME algorithm. We aim at traducing the three criteria of mutual exclusion in regular alternation-free μ-calculus, and verifying the correctness of the C implementations *w.r.t.* these properties.

A number of steps will have to be performed to compute a reduced model before evaluating it, and these are described below.

```
<?xml version="1.0" encoding="UTF-8"?>
<api name="sharedmemory" includeheader="sharedmemory.h"
     include="sharedmemory.c" folder="/usr/share/LTStool/apis/sharedmemory/">
<globalvar name="svsm" type="shared_structure">
</globalvar>
<initialization name="initialize_shared_mem_structures(CAESAR_S)">
</initialization>
<functions>

<function name="label" sname="LABEL" type="1" static="true">
<arg name="value" typeArg="1" argref="0" type="char" argsize="100"
     labelname="value">
</arg>
</function>

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

<function name="screate" sname="create_shared_memory" type="1">
<arg typeArg="1" argref="0" type="char" argsize="20"
     labelname="reg_name"/>
<arg typeArg="1" argref="1" type="int" labelname="size"/>
<arg typeArg="1" argref="2" type="void *" labeltype="int"
     labelname="initial"/>
<arg typeArg="1" argref="2" type="int" labeltype="char" labelsize="20"
     returned="true"/>
</function>

<function name="swrite" sname="write_shared_memory" type="1">
<arg typeArg="1" argref="0" type="int" labeltype="char" labelsize="20"
     labelname="desc"/>
<arg typeArg="1" argref="1" type="void *"  labeltype="int"
     labelname="segment"/>
<arg typeArg="1" argref="2" type="int" labelname="size"/>
<arg typeArg="1" argref="3" type="int" returned="true"/>
</function>

<function name="sclose" sname="close_shared_memory" type="1">
<arg typeArg="1" argref="0" type="int" labeltype="char" labelsize="20"
     labelname="desc"/>
<arg typeArg="1" argref="1" type="int" returned="true"/>
</function>

</functions>
</api>
```

Fig. 3. XML file describing the abstract shared memory API model

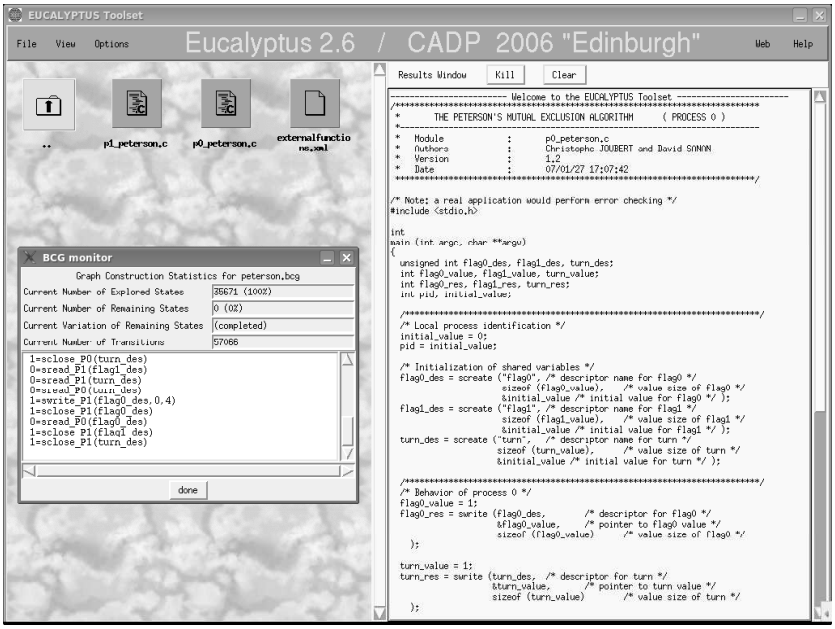

Fig. 4. Screen snapshot of EUCALYPTUS executing the state space generation.

In the demonstration we will not initiate the individual steps needed to compute these quantities. Instead we shall launch an SVL *script [2] (from the GUI) that carries out these steps. The script will be explained to the audience step-by-step (see following demo.svl script) .*

```
(*===========================================================================*)
(*                                  demo.svl                                 *)
(*===========================================================================*)

(*
 * This SVL script performs the following activities:
 * (1)  program analysis using C.OPEN and ANNOTATOR of "p0_peterson.c"
 *      and "p1_peterson.c"
 * (2)  program slicing of "p0_peterson.c" and "p1_peterson.c" using C.OPEN
 *      and GENERATOR; this results in an LTS file named "p1_p0.bcg" which
 *      size information is displayed.
 * (2') minimization modulo strong bisimulation of the resulting automaton
 *      using BCG_MIN; this results in an LTS file named "peterson.bcg"
 *      which size information is displayed.
```

```
 * (3)  verification of the temporal formulas defined in prop.mcl on
 *      "peterson.bcg" using EVALUATOR and BCG_OPEN
 *)

(* program analysis w.r.t. api influence analysis *)                    (*1*)
% c.open -static -filelist 1 p0_peterson.c 1 annotator.a -api\
%                -xml p0_peterson-static.xml
% c.open -static -filelist 1 p1_peterson.c 1 annotator.a -api\
%                -xml p1_peterson-static.xml

(* program slicing using the program analysis result *)                (*2*)
% c.open -filelist 2 p1_peterson.c 1 p0_peterson.c 1 generator\
%        -monitor p1_p0.bcg

(* display size of the LTS *)
% bcg_info "p1_p0.bcg"

(* minimization modulo strong bisimulation *)                          (*2'*)
"peterson.bcg" = strong reduction with bcg_min of "p1_p0.bcg";

(* display size of the LTS *)
% bcg_info "peterson.bcg"

% echo
% echo "-------------------------------------------------------------"
% echo "| Checking the temporal logics properties defined in prop.mcl |"
% echo "| on peterson.bcg using EVALUATOR                           |"
% echo "-------------------------------------------------------------"
verify "prop.mcl" in "peterson.bcg";                                   (*3*)

(* cleanup *)
% SVL_RECORD_FOR_CLEAN "peterson.bcg"
% SVL_RECORD_FOR_CLEAN "p1_p0.bcg"
% SVL_RECORD_FOR_CLEAN "p1_peterson-static.xml"
% SVL_RECORD_FOR_CLEAN "p1_peterson.xml"
% SVL_RECORD_FOR_CLEAN "p0_peterson-static.xml"
% SVL_RECORD_FOR_CLEAN "p0_peterson.xml"
% SVL_RECORD_FOR_CLEAN "functions.h"
% SVL_RECORD_FOR_CLEAN "graph.c"
% SVL_RECORD_FOR_CLEAN "graph.o"
% SVL_RECORD_FOR_CLEAN "header.h"
% SVL_RECORD_FOR_CLEAN "trans.m"
% SVL_RECORD_FOR_CLEAN "trans.t"
% SVL_RECORD_FOR_CLEAN "annotator"
% SVL_RECORD_FOR_CLEAN "generator"
% SVL_RECORD_FOR_CLEAN "generator.o"
```

The study of the Pme can be divided into three major steps parts.

1. analyse P0 and P1 programs modulo Api influence analysis,

2. slice the PME program using analysis result of step 1, and obtain a minimized PME model,
3. evaluate the three criteria on the model.

The first point (in line `(*1*)` of the script) is achieved using both new tools, C.OPEN and ANNOTATOR, on each C program describing process P0 and P1. Within the SVL-script, this is done calling C.OPEN with the following options:

- *-static* indicates C.OPEN to generate a control flow graph (20 states, 20 transitions) out of the C input program, and
- *-filelist 1 p0_peterson.c 1* specifies that only one C program, called *p0_peterson.c* here, and one instantiation of it will be processed.

The option *-api* of ANNOTATOR indicates that API influence analysis, *i.e.* an influence analysis [3, 4] that preserves API variables, will be computed. Choosing option *-xml p0_peterson-static.xml* puts the analysis result in an XML file. For each program control point, ANNOTATOR gives a list of variables influencing the PME abstract model. This tool further detects that *pid* is the only variable that does not influence any control point of the PME program, and thus it can be safely ignored for the verification of the protocol.

The control flow graph of P0 will be displayed to the audience
using the BCG_EDIT *tool (cf. Figure 5),*
as well as the XML *influence analysis results.*

The second point, slicing parts of P0 and P1 programs that read or modify variable *pid*, is done by C.OPEN using the XML influence analysis result files from ANNOTATOR, together with the GENERATOR application (in line `(*2*)` of the script), which successfully constructs a smaller LTS than the one previously computed without slicing. The obtained LTS has 583 states and 1 052 transitions, which is more than 20% smaller than the original state space without reduction. This LTS model could serve as an immediate input to the model checking tool, but we decided to add another reduction step (line `(*2'*)` of the script). Calling the strong option of the BCG_MIN reductor, we build a minimized LTS modulo strong bisimulation. This LTS has 268 states and 502 transitions.

The third step, verifying each one of the three mutual exclusion criteria is done by the EVALUATOR model checker (in line `(*3*)` of the script), which evaluates the various temporal properties on the minimized model. Here we use a file, called *prop.mcl*, that defines the set of properties as regular alternation-free μ-calculus formulas. The content of the file is described in the sequel.

The three regular alternation-free μ-calculus formulas,
will be displayed and explained to the audience
using the EUCALYPTUS *interface of* CADP,
as well as the verification results produced by EVALUATOR.

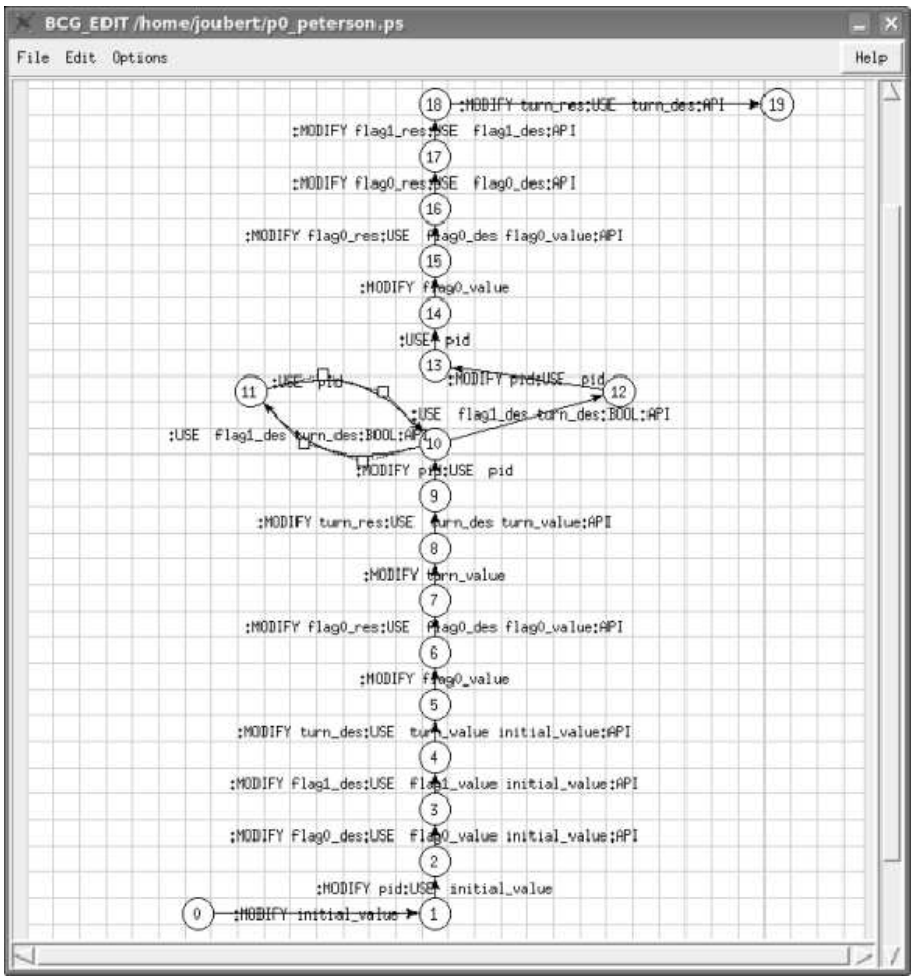

Fig. 5. CFG for process 0 of the Peterson's mutual exclusion implementation.

```
(*=====================================================================*)
(*                                prop.mcl                             *)
(*                ( based on spec.mcl from CADP demo_21 )              *)
(*=====================================================================*)

(* Basic predicates over actions *)

(* Request access to the critical sections *)
macro REQ0 () = '.*swrite_P0(flag0_des,1,.*' end_macro
macro REQ1 () = '.*swrite_P1(flag1_des,1.*' end_macro

(* Release the critical sections *)
macro REL0 () = '.*swrite_P0(flag0_des,0,.*' end_macro
macro REL1 () = '.*swrite_P1(flag1_des,0.*' end_macro

(* Read the variables *)
macro REQ0_RT () = "1=sread_P1(flag0_des)" end_macro
macro REQ0_RF () = "0=sread_P1(flag0_des)" end_macro
macro REQ1_RT () = "1=sread_P0(flag1_des)" end_macro
macro REQ1_RF () = "0=sread_P0(flag1_des)" end_macro
macro TURN_R1 () = '1=sread_P1(turn_des)' end_macro
macro TURN_R0 () = '0=sread_P0(turn_des)' end_macro

(* Critical sections *)
macro CS0 () = (REQ1_RF or TURN_R0) end_macro
macro CS1 () = (REQ0_RF or TURN_R1) end_macro

(*=====================================================================*)
(* Temporal properties for three mutual exclusion criterias *)

(*---------------------------------------------------------------------*)
(*
 * Mutual exclusion. After Pi has entered its critical section CSi,
 * it is not possible that Pj enters its critical section CSj until
 * Pi has released CSi
 *)

[ true* . ((CS0 . (not REL0)* . CS1) | (CS1 . (not REL1)* . CS0)) ] false

and
(*---------------------------------------------------------------------*)
(*
 * Progress requirement. Reachability of Pi's critical section if Pj does
 * not request the access.
 * After Pi has requested the CSi and after Pj has released CSj, all paths
 * lead to an access of Pi (CSi).
 *)

[ true* ] (
    [ REQ0 . (not CS0)* . REL1 ] mu X . (< true > true
```

```
                              and [ not CS0 ] X)
    and
    [ REQ1 . (not CS1)* . REL0 ] mu X . (< true > true
                              and [ not CS1 ] X)
)

and
(*-------------------------------------------------------------------------*)
(*
 * Bounded waiting: Reachability of both critical sections under
 * fair scheduling of actions
 *)

[ (not (CS0 or CS1))* ] (
    [ (not CS0)* ] < (not CS0)* . CS0 > true
    and
    [ (not CS1)* ] < (not CS1)* . CS1 > true
)
```

The satisfaction of the properties indicates that our implementation of the Pme algorithm preserves the three criteria of mutual exclusion, *w.r.t.* our abstract shared memory Api model.
Finally, we can notice that the implicit Lts model of the Pme algorithm can also be used as input for other Cadp verification tools (*e.g.*, for bisimulation, simulation, testing, *etc.*), to get various other insights into the system.

If time permits, we will further demonstrate how existing tools from Cadp *can be directly used with* C.Open*, and we will hint, for example, at the interactive execution of the* Pme *algorithm with the* Simulator *application illustrating the critical section access from both processes.*

B Tool particularities

The last stable release of Cadp, called Cadp 2006 "Edinburgh", can be obtained free of charge for academic purposes. Institutes must sign a license agreement with Inria. Cadp is available for the following platforms:

- Sun stations (Sparc processors) running Solaris 8 or higher with a C compiler (such as Sun Studio 11 "cc" compiler or the GNU "gcc") and the Ghostview software,
- PC computers (i386-like processors) running Linux (kernel version $\geq$ 2.4, glibc version $\geq$ 2.3),
- PC computers (i386-like processors) running Windows 2000 or XP with the Cygwin and Ghostscript/Gsview software installed, and

– Apple computers (PowerPC processors) running Mac OS X 10.2 or higher with the X11 and Ghostscript/Gsview software installed.

The installation of needed specific software package is triggered by the installation procedure for CADP. See http://www.inrialpes.fr/vasy/cadp/ for further information.

The C.OPEN tool component is available for the same platforms as CADP, and ANNOTATOR for Solaris and Linux stations. They can be downloaded free of charge from http://www.lcc.uma.es/gisum/tools/smc/ as an add-on to CADP. The BCG_MIN, GENERATOR, EVALUATOR, SIMULATOR, SVL, and EUCALYPTUS tools are already part of the current CADP distribution.

References

1. Raynal, M.: Algorithmique du parallelisme : le probleme de l'exclusion mutuelle. (1984)
2. Garavel, H., Lang, F.: SVL: a scripting language for compositional verification. In Kim, M., Chin, B., Kang, S., Lee, D., eds.: Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea), IFIP, Kluwer Academic Publishers (2001) 377–392 Full version available as INRIA Research Report RR-4223.
3. Cámara, P., Gallardo, M., Merino, P.: Abstract matching for software model checking. In Valmari, A., ed.: Proceedings of the 13th International SPIN Workshop on Model Checking of Software SPIN'06 (Vienna, Austria). Volume 3925 of Lecture Notes in Computer Science., Springer Verlag (2006) 182–200
4. Gallardo, M., Joubert, C., Merino, P., Sanán, D.: On-the-fly API influence analysis of software. In Merino, P., Bakkali, M., eds: Proceedings of the 2nd International Conference on Science and Technology JICT'2007 (Málaga, Spain), Spicum (2007)