On-the-fly model checking for C programs with extended CADP in FMICS-jETI *

María del Mar Gallardo, Pedro Merino
and David Sanán
University of Málaga
ETSI Informática
Campus de Teatinos s/n, 29071 Málaga, Spain
{gallardo,merino,sanan}@lcc.uma.es

Christophe Joubert
Technical University of Valencia
DSIC / ELP
Camino de Vera, s/n, 46022 Valencia, Spain
joubert@dsic.upv.es

Abstract

A current trend in the software engineering community is to integrate different tools in a friendly and powerful development environment for use by final users. This is also the case for tools based on formal methods, which are very valuable for increasing confidence in the reliability of software. This paper contributes to one promising approach to make this integration possible, the project FMICS-jETI. *This project aims to obtain an active repository of tools based on formal methods in such a way that users can access and combine all the tools simply by defining a graph with the tools and the files they manage. In particular, the paper explains how two new modules of the well known toolset CADP are added to* FMICS-jETI. *These new modules, named* C.OPEN *and* ANNOTATOR *extend* CADP *with functions to manage* C *programs in this toolset.*

1. Introduction

Current software systems are becoming more concurrent, distributed and pervasive and their complexity requires that many tools be used during development. For instance, in order to analyse the software properties, checking compiler errors is only the starting point, and tools with more powerful functionalities are necessary, like reliability or performance analysers. In this context, the formal methods community has produced a number of tools that manage specification and programming languages. In particular, we have participated in the development of several tools for software reliability which combines formal techniques like static analysis, abstract interpretation and model checking (see ASPIN [7], ANNOTATOR [4, 5], C.OPEN [8]). It is clear that the combination of existing tools can be a way to obtain powerful environments for software developers, who are willing to accept new tools provided that they are not forced to learn new languages or to write new specifications.

Considering the problem of tool integrations, a first approach is to define intermediate languages, which are used as the transfer notation between the tools to be integrated. These languages are not ad-hoc or user-oriented, but they suitable for parsers and for algorithms. This approach was originally followed by the SPECS [19] and SEDOS [3] projects and, more recently in IF [1] Veritech, [14], BANDERA [12] and in the toolset CADP [10]. Another related approach to integrating tools is the use of XML [6] which has recently produced the definition and implementation of the intermediate language of interchange PiXL [7].

A different approach to tool integration is the construction of integrated environments to manage a group of tools, using internal translators between the source and the destination tool. The ETI platform [20] was designed with this approach. In ETI, coordination among tools is obtained through the definition of functional taxonomies that each tool exports when it is integrated into the platform. Thus, the environment is able to recognize common or compatible functionality. ETI has been recently redesigned and is now FMICS-jETI [15, 16]. Now, the project FMICS-jETI is promoting the use of this platform to integrate formal methods tools [2, 21].

In this paper, we show how to include two of our tools, C.OPEN and ANNOTATOR, into FMICS-jETI. C.OPEN is designed to translate C programs into the internal notation of CADP. ANNOTATOR takes the output of C.OPEN and perform static analysis in order to obtain information to optimize further processing with other tools in CADP. Both tools are now being used to do model checking of C programs. The successful integration of both tools in FMICS-jETI is an important step towards showing that other modules of CADP can also be integrated.

The paper is organized as follows. Section 2 describes

*Work partially supported by TIN2004-7943-C04-01 and TIN 2005-09405-C02-01

the C.OPEN and ANNOTATOR tools with the standard implementation of CADP. Section 3 contains the details on how to include both tools in FMICS-JETI. Section 4 presents a case study with the resulting integration. Finally, we present some conclusions in Section 5.

2. Extending CADP for C code

This section presents the new tools developed to extend CADP to manage C code.

2.1. CADP overview

CADP can be considered a traditional toolbox for the analysis of communication protocols.

Using a modular architecture, CADP includes compilers to translate several input formalisms (LOTOS, BCG) into a generic format (an LTS). The different applications of CADP uses the LTS as an internal representation of the input language.

There is a wide variety of tools providing different fuctionailities in CADP. For example, it contains a module to analyze whether two specifications 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 contains some tools that could be 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).

Nevertheless, 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. Applications in OPEN/CÆSAR have the functionality separated into three different modules: the graph, the storage and the exploration modules. The main functionality is carried out by the exploration module and it handles the other two modules. OPEN/CÆSAR provides a set of libraries with the necessary structures to store the labels and states of the LTS. These structures constitute the storage module. 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.

However, CADP does not support programming languages in a native way, it only provides compilation for some formalism such as LOTOS or binary code graphs BCG through CAESAR.OPEN and BCG_OPEN. We extend CADP with C.OPEN, based on [8], making it possible to use the whole environment with C programs. A second extension is ANNOTATOR [4] that allows us to perform static analysis in order to improve the processing with other modules in CADP(see fig. 1).

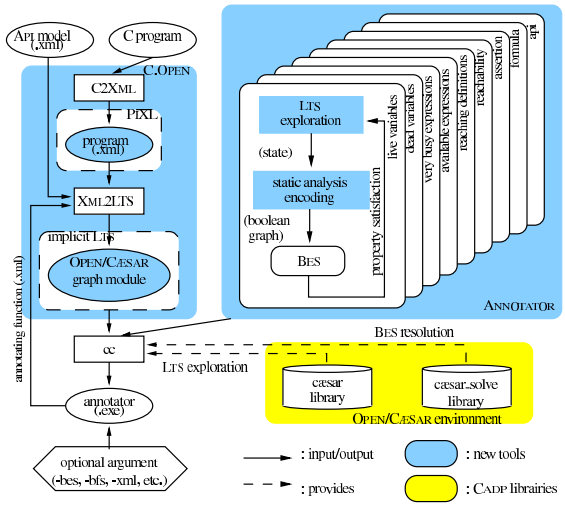

Figure 1. C.OPEN and ANNOTATOR tools

2.2. C.OPEN

C.OPEN adds the possibility of using the CADP framework to support C program input. The idea that the OPEN/CÆSAR [9] environment could be connected to a C compiler and that existing CADP tools can therefore be extended to this new class of specification can be considered an important step towards reusing well-stablished verification toolboxes. In particular, C.OPEN deals with ¸code that makes use of well-defined APIs; its main purpose is to test the correct use of such APIs using the whole CADP environment. C.OPEN takes as inputs a system described by a set of C programs, an Apis model operating system written in C, the translation rules for the Api's model represented in XML, and an OPEN/CÆSAR application (e.g., ANNOTATOR). In the translation task of generating the OPEN/CÆSAR graph module (see sub-section above), C.OPEN is divided in several subtasks that are carried out by different modules. The task of handling and managing the C code is not trivial due to the complexity of this language. So, it is translated into XML, and more specifically, into a PIXL scheme [7] with a tool called C2Xml. Another tool called Xml2LTS analyzes the code to stablish the external calls to the language, to perform an

influence analysis of variables and to generate the necessary structures and transitions, *w.r.t.*the API used in the input code, that forms the final LTS. In the code analysis, users can decide whether the influence analysis of variables is performed by C.OPEN or supported by other tool such as ANNOTATOR which provides a more strict analysis. Once the LTS from the initial system input has been generated, C.OPEN compiles the generated LTS with the given exploration module and the storage module (as an OPEN/CÆSAR library) and it executes the resulting application. It currently offers the possibility of generating either a control flow graph (Cfg), to be used by ANNOTATOR , or an explicit state space of a program as an implicit LTS

To properly validate the input system *w.r.t.* some API, C.OPEN needs a model of the behaviour for the external calls appearing in the program and an XML file with the rules for translating the external API functions into the modelled one. Table 1 shows, as an example, the API `Shared Memory` that provides four basic functions for dealing with a shared resource, that is, `create`, `read`, `write` and `close`. We can consider the shared memory as a composition of several regions with an unique name and size. New regions are created through the `screate` function model. It receives the name of the new region, and its size and initial value and it ensures that no region has been previously created with the same name, size or 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 to a non-existent region returns an error code.

C.OPEN uses the so-called *translation rules* to properly transform each external function. Possibly, external function call parameters will be modified in the corresponding model. Therefore, the way to translate each call is given in an XML file where, for each function call, the arguments that must be preserved or that must be added into 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. Moreover, the arguments of functions may have a different representation in the label of the LTS, so, it is necessary to specify this aspect in the translation rules file. This is the case in the first argument of the `sread`, the LTS uses the name of the variable instead of the real variable value, this is done setting the option `varname` to `yes`.

```
<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>
```

Figure 2. sread translation rules

2.3. ANNOTATOR

ANNOTATOR implements standard data flow analysis algorithms on a CFG, by using boolean equation systems (BESs) [4]. It also computes various influence analyses [10], 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 be carried out and an LTS describing the CFG of a program, in which instructions are abstracted to the strictly necessary information (i.e., modified and defined variables, used expressions 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 [11]. 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 [17] library, which offers several on-the-fly resolution algorithms, based on different search strategies (e.g., breadth-first). Depending on the option selected by the user, the analysis result is written as 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. Figure 1 shows how the generation of the Cfg from a program with C.OPEN lets us to use ANNOTATOR to determine which variables in this program must be verified with CADP.

ANNOTATOR consists of two parts: a front-end, responsible for encoding the static analysis of LTS as a (parameterised) BES resolution, and a back-end, responsible for (parameterised)Bes resolution, playing the role of verification engine. Back-end is obtained by using algorithms of the CÆSAR_SOLVE library. Globally, the approach to on-the-fly static analysis is both to construct the LTS and corresponding (parameterised) LTS on the fly and to determine the final value of boolean variables of interest. Only the

```
#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 */ );
  /*************************************************************************/
}
```

Figure 3. peterson mutual exclusion code

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)		

Table 1. Shared Memory API functions

part of both graphs where it necessary to perform the static analysis is explored incrementally.

2.4 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 will be used together with a model of this API. We will show how C.OPEN works implementing the Peterson's mutual exclusion (PME) algorithm and using several CADP tools as generator, simulator or evaluator to check the correctness of the programs.

PME is a concurrent programming algorithm for mutual exclusion that allows only 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 the 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 system to be analyzed is composed of two programs, p0_peterson.c (figure 3) and p1_peterson.c, which use the Peterson mutual exclusion algorithm for accessing to a common critical section. Both programs are symmetrical, they differ only in the program pid, and in the control flag variables that guard the critical section. The programs are structured as follows: First, the different shared variables controlling the critical section are created; then, and before going into the critical section, both processes make an active wait for the critical section; thirdly, each process updates the flag shared variable to ensure that the other process can not exit from the active wait; finally the shared structures are closed and the programs finish.

Figure 4 shows the execution of C.OPEN with GENERATOR . The command line C.Open takes as arguments the input for C.OPEN (typically the C system) and the exploration module, GENERATOR in this example, with the corresponding parameters (i.e. the file where GENERATOR will save the generated BCG). In the example, C.OPEN generates and invokes the executable for GENERATOR.

```
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''
```

Figure 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)
```

Figure 5. Information of the explicit LTS generated by generator

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, making it easier to manage. For that purpose 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-compression tau-confluence, tau*.a equivalence, safety equivalence, trace equivalence, or weak trace equivalence). Figure 6 shows the application of REDUCTOR to the BCG previously obtained. Therefore, if we apply REDUCTOR to the bcg obtained after applying GENERATOR with a total reduction, we obtain a smaller LTS equivalent with only 157 different states and 288 transitions.

To check the correctness of the system being evaluated,

```
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
```

Figure 6. Information about the explicit LTS after being reduced with reductor

we test that 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* is false or *turn* is 0. Either case, 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 entering 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.

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

- Analysis of P0 and P1 programs modulo API influence analysis. For this, we use C.OPEN next to ANNOTATOR to determine the variables that preserve the API used. For each program control point, ANNOTATOR gives a list of variables influencing the PME abstract model and puts the analysis results in an XML file. 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 protocol verification.
- The PME program is sliced using the analysis result of step 1, and a minimized PME model is obtained. This is done by C.OPEN using the XML influence analysis result files from ANNOTATOR, together with the Generator application, which successfully constructs a smaller LTS than the one previously computed without slicing.
- We use EVALUATOR, a model checker which evaluates the various temporal properties in the minimized model The satisfaction of the properties indicates that

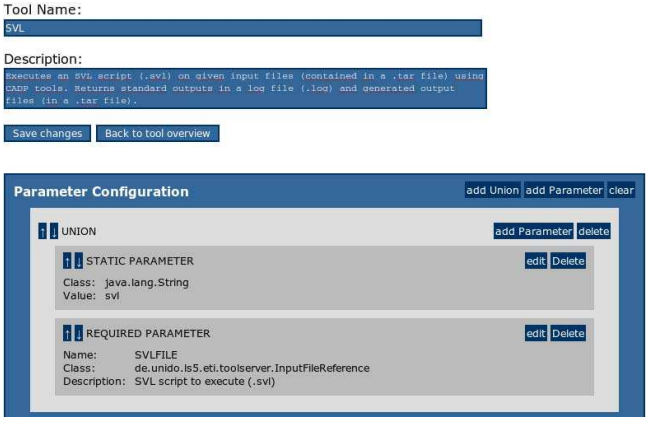

Figure 7. Snapshot of the jETI web configurator for SVL SIB

our implementation of the PME algorithm preserves the three mutual exclusion criteria, *w.r.t.* our abstract shared memory API model.

3. Adding C model checking tools to FMICS-jETI

3.1. FMICS-jETI architecture

FMICS-JETI is based on a client/server architecture. Through a client application (JABC), it is possible to define graphs for the integration of different tools and their execution on a remote server. For adding new services into FMICS-JETI, it offers an easy and handy interface, HTML tool configurator. This configurator allows automatic SIBs (Service-Independent Building Block) generation from the specification of the parameters for such service. SIBs are java classes that are called on the client side and are responsible for the communication with the server which allocates the tool associated to the service provided by the SIB

The JABC client is a graphical interface for the specification of SLG graphs. These graphs allow the combination of the different remote tools, and see them as sequential programs.

3.2. SVL SIB on jETI server

To remotely execute C.OPEN and ANNOTATOR, we have designed only one SIB, SVL SIB. This SIB adds a service that remotely executes a SVL (Script Verification Language) script which is responsible for calling to C.OPEN and ANNOTATOR with all the necessary parameters to invoke them, such the C input files or the configuration ones.

```
<etitoolserver serverURI='http://d3.lcc.uma.es:8080/services/ETI'>

<tool name='SVL' active='true' class='de.unido.ls5.eti.toolserver.RuntimeUnix' method='exec'>
     <description>Executes an SVL script (.svl) on given input files (contained in a .tar file) using CADP tools.
     Returns standard outputs in a log file (.log) and generated output files (in a .tar file).</description>
     <array class='java.lang.Object'>
     <union>
          <parameter class='java.lang.String' value='svl' />
          <parameter name='SVLFILE' class='de.unido.ls5.eti.toolserver.InputFileReference' required='true'
          description='SVL script to execute (.svl)' />
     </union>
     <parameter name='INFILE' class='de.unido.ls5.eti.toolserver.InputFileReference' required='true'
     description='Archive (.tar) of input specification files' />
     <union>
          <parameter class='java.lang.String' value='>' />
          <parameter name='LOGFILE' class='de.unido.ls5.eti.toolserver.OutputFileReference' required='true'
          description='File (.log) to write log information to' />
     </union>
     <parameter name='OUTFILE' class='de.unido.ls5.eti.toolserver.OutputFileReference' required='true'
     description='File (.tar) to write all generated output files to' />
     </array>
</tool>

</etitoolserver>
```

Figure 8. XML file describing the new SVL jETI-SIB

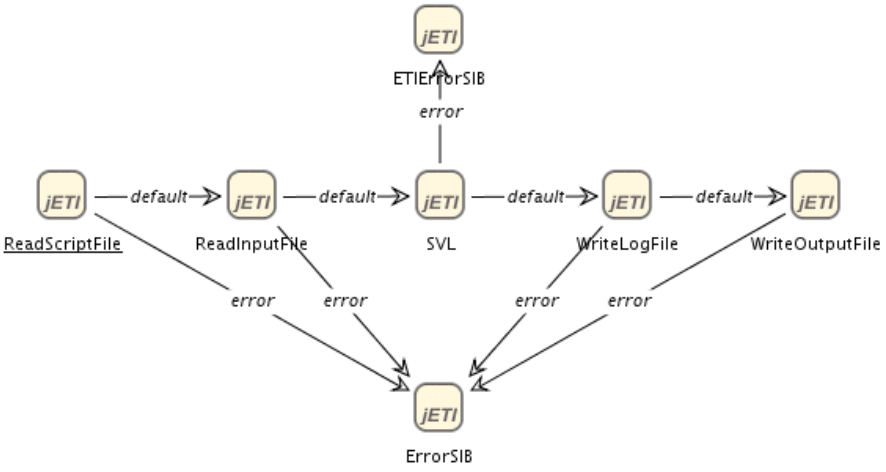

Figure 9. Service logic graph of the Peterson's mutual exclusion case-study

Using SVL as a service let us simplify the integration of CADP with FMICS-JETI. Moreover, defining a SVL script instead of creating different SIBs, one for each CADP tool, lets us use the whole set of CADP tools adding a unique service. The SVL SIB invoques the SVL interpreter on the server side and it requires four parameters, two of them are input parameters and the others are from output. Therefore, it needs the SVL script containing the necessaries commands (typically, file manipulation and CADP tools) for the verification process that will be executed in the server. The system being analyzed, as well as configuration files for the tools involved in the verification process, are passed to the script through the second input argument of the SVL SIB. The different files are packed into a tar file, and the script, when it is executed in the server, untars the files enabling the script to work correctly.

The output arguments of the SIB are, on one hand, the execution log of the script, containing the output of the different tool. On the other, the SIB returns the different files generated by the tools invoked remotely.

4. The Peterson's mutual exclusion case-study

The Peterson's mutual exclusion (PME) protocol is 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 algorithm taken from [18] for mutual exclusion between two processes (figure 3).

4.1. Service logic graph on jABC client

In [13], the authors gave a C implementation of this protocol as well as an SVL script describing the whole verification process of the PME protocol with standalone CADP. In order to realize the same experimentation through the FMICS-JETI platform, it is necessary to define a *service logic graph* (SLG) in the JABC client, which invokes for each SIB composing the graph, the corresponding remote tool on a specific FMICS-JETIserver. Hence, once we defined our SVL SIB and made it available on our FMICS-JETI server, we need to use it in a JABC graph. Figure 9 gives an SLGthat first reads two files, one being the SVL script (*demo.svl*) and the other being an archive of all input files (*demo_41_input.tar*). The SVL SIB is then called with those two arguments and, upon success, returns two files, one for the remote standard output trace (*stdout.log*) and one for the files generated during the experiment (*demo_41_output.tar*). If an error occurs, it is either treated locally(*ErrorSIB*), or remotely (*ETIErrorSIB*).

This methodology enables the remote use of C.OPEN and ANNOTATOR tools, which are C model checking extensions of the last stable release of the CADP toolbox, without needing a local installation these tools. This approach is even more attractive since our SVL SIB file, and FMICS-JETI server enable the execution of all verification tools as well as all 40 SVL demos available part of the current CADP toolbox. This is a great advantage when one wants to quickly test the applicability of an approach using state-of-the-art up-to-date technologies and tools, without the necessity of dealing with hardware, software and license dependencies.

Using the *stdout.log* file generated by the execution of our SLG we observe that the evaluation of three alternation-free μ-calculus formulas by the EVALUATOR model checker returns true. The satisfaction of these properties indicates that our C implementation of the PME algorithm preserves the three criteria of mutual exclusion, *w.r.t.* our abstract shared memory API model.

5. Conclusions

We have presented another case study in the evolution of the project FMICS-JETI. In particular, we have integrated the tools C.OPEN and ANNOTATOR and we have constructed the SIB and the SLG to access these tools from the JABC client. We also present an example of how to use the new tools within this framework. The success in bringing the new tools to this environment in a very short time proves that most (maybe all) the components of CADP can be also integrated into the FMICS-JETI project promoted by the ERCIM Working Group on Formal Methods for Industrial Critical Systems (FMICS) (see information at http://www.inrialpes.fr/vasy/fmics/).

References

[1] M. Bozga, J. Fernandez, L. Ghirvu, S. Graf, and L. Krimm, J.and Mounier. If: A validation environment for timed asynchronous systems. In *Proceedings of CAV'00*, volume 1855, 2000.

[2] L. Brim and M. Leucker. Parallel model checking and the fmics-jeti platform. 2007. To appear in Proc. of ICECCS07.

[3] M. Diaz, C. Vissers, and J. Ansart. Sedos software environment for the design of open distributed systems. In *The formal Description Technique LOTOS*. North-Holland, 1989.

[4] M. Gallardo, C. Joubert, and P. Merino. Implementing influence analysis using parameterised boolean equation systems. In N. Halbwachs and L. 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.

[5] M. Gallardo, C. Joubert, and P. Merino. On-the-fly data flow analysis based on verification technology. In R. Drechsler, S. Glesner, and J. Knoop, editors, *Proceedings of the*

6th International Workshop on Compiler Optimization meets Compiler Verification COCV'2007 (Braga, Portugal). Elsevier, March 2007.

[6] M. Gallardo, J. Martinez, P. Merino, and E. Pimentel. A tool for abstraction in model checking. *Software Tools for Technology Transfer*, 5, 2004.

[7] M. Gallardo, J. Martnez, P. Merino, P. Nuez, and E. Pimentel. Pixl: Applying xml standards to support the integration of analysis tools for protocols. *Science of Computer Programming*, 65:57–69, March 2007.

[8] M. Gallardo, P. Merino, and D. Sanan. Towards model checking c code with open/caesar. In *Proc. of MSVVEIS'06*, pages 198–201, 2006.

[9] H. Garavel. Open/caesar: An open software architecture for verification, simulation, and testing. In B. Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98,*, 1998.

[10] H. Garavel, F. Lang, and R. Mateescu. Cadp 2006: A toolbox for the construction and analysis of distributed processes. *In Proc. of CAV'07*, To appear.

[11] H. Garavel and W. Serwe. State space reduction for process algebra specifications. *Theor. Comput. Sci.*, 351(2):131–145, 2006.

[12] J. Hatcliff, M. Dwyer, C. Pasareanu, and Robby. Foundations of the bandera abstraction tools. In *The Essence of Computation*, volume 2566. LNCS, 2003.

[13] http://www.lcc.uma.es/gisum/tools/smc. *C.OPEN and ANNOTATOR: Tools for On-the-Fly Model Checking C Programs*. UMA/GISUM and UPV/ELP, 2007.

[14] S. Katz. Faithful translations among models and specifications. In *Proc. of Formal Methods Europe*, 2001.

[15] T. Margaria, R. Nagel, and B. B. Steffen. Remote integration and coordination of verification tools in jeti. 2005.

[16] T. Margaria and B. Steffen. Advances in the fmics-jeti platform for program verification. 2007. To appear in Proc. of ICECCS07.

[17] R. Mateescu. Caesar solve: A generic library for on-the-fly resolution of alternationfree boolean equation systems. *Springer Int. J. on Soft. Tools for Tech. Trans.(STTT)*, 8(1):37–56, 2006.

[18] M. Raynal. *Algorithmique du parallelisme : le probleme de l'exclusion mutuelle*. 1984.

[19] W. Reed, R.and Bouma, J. Evans, M. Dauphin, and M. Michel. The specs consortium. specification and programming environment for communication software. North-Holland, 1993.

[20] B. Steffen, T. Margaria, and V. Braun. The electronic tool integration platform: concepts and design. *Special section on the Electronic Tool Integration Platform, Int. Journal on SoftwareTools for Technology Transfer*, 1, 1997.

[21] B. Steffen, T. Margaria, and L. M. The learnlib in fmics-jeti. 2007. To appear in Proc. of ICECCS07.