Science of Computer Programming 77 (2012) 375–392

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A model-extraction approach to verifying concurrent C programs with CADP☆

M.M. Gallardo [a], C. Joubert [b,*], P. Merino [a], D. Sanán [a]

[a] Universidad de Málaga, Spain

[b] Prodevelop/Universitat Politècnica de València, Spain

ARTICLE INFO

Article history:
Received 13 August 2010
Received in revised form 1 October 2011
Accepted 3 October 2011
Available online 19 October 2011

Keywords:
Software model checking
Data flow analysis
Labeled transition system
Boolean equation system
CADP verification toolbox

ABSTRACT

The development of reliable software for industrial critical systems benefits from the use of formal models and verification tools for detecting and correcting errors as early as possible. Ideally, with a complete model-based methodology, the formal models should be the starting point to obtain the final reliable code and the verification step should be done over the high-level models. However, this is not the case for many projects, especially when integrating existing code. In this paper, we describe an approach to verify concurrent C code by automatically extracting a high-level formal model that is suitable for analysis with existing tools. The basic components of our approach are: (1) a method to construct a labeled transition system from the source code, that takes flow control and interaction among processes into account; (2) a modeling scheme of the behavior that is external to the program, namely the functionality provided by the operating system; (3) the use of demand-driven static analyses to make a further abstraction of the program, thus saving time and memory during its verification. The whole proposal has been implemented as an extension of the CADP toolbox, which already provides a variety of analysis modules for several input languages using labeled transition systems as the core model. The approach taken fits well within the existing architecture of CADP which does not need to be altered to enable C program verification. We illustrate the use of the extended CADP toolbox by considering examples of the VLTS benchmark suite and C implementations of various concurrent programs.

Published by Elsevier B.V.

1. Introduction

Static analysis [1] and model-checking [2] techniques have been widely used over the last 30 years to help prove the correctness of systems and to find critical errors in their design. In recent years, the use of model-checking techniques on program source code, which is commonly known as *software model checking*, has become a promising way for the automatic verification of software, demonstrating that programs are reliable with regard to a specification. Software model checking aims to make it easy for engineers to use powerful traditional model-checking techniques without needing to manually convert the source code of a program into a formal high-level specification language such as Petri nets, SDL, or *labeled transition systems* (LTSs). Therefore, since it can be used by non-expert users, software model checking appears to be an excellent way to prove system correctness, especially when the project involves existing code. This paper presents novel contributions to the combination of static analysis and software model checking for concurrent C programs.

☆ This work has been supported by the Spanish MICINN under grants TIN2007-68093-C02, TIN2010-21062-C02-02, FEDER, the MICINN INNCORPORA-PTQ program, and the Generalitat Valenciana under grants Emergentes gv/2009/024 (TAAS) and PROMETEO2011/052.

* Corresponding author.
E-mail addresses: gallardo@lcc.uma.es (M.M. Gallardo), joubert@dsic.upv.es (C. Joubert), pedro@lcc.uma.es (P. Merino), sanan@lcc.uma.es (D. Sanán).

0167-6423/$ – see front matter. Published by Elsevier B.V.
doi:10.1016/j.scico.2011.10.003

Automatic verification of C code has been addressed in many previous proposals. Some of them are based on the construction of model checkers for C from scratch, like SLAM [3,4], BLAST [5], or COCCINELLE [6]. A different and appealing approach is to reuse existing model-oriented, model checking tools by constructing model extractors. That approach is based on producing a high-level model that can benefit from existing tools for verification, simulation, visualization and other model-based processing capabilities. Some standard examples are MODEX [7], BANDERA [8], or JPF [9].

In the present work, we aim to leverage the capabilities of the *Construction and Analysis of Distributed Processes* (CADP [10]) toolbox to verify concurrent C programs based on model extraction. CADP plays a unique role in the formal verification community. It offers an open environment for both *on-the-fly*, *e.g.*, demand-driven, and *global* explicit verification of asynchronous distributed hardware and software systems with interleaving concurrency. Such systems are internally described by timeless labeled transition systems and modal mu-calculus properties, *i.e.*, branching and linear time logical properties. Verification runs are carried out using parallel computers as well as either breadth-first or depth-first search strategies. CADP provides a software framework called OPEN/CÆSAR [11] to easily connect new languages and compilers to CADP verification tools and to connect new verification functionalities to all CADP language inputs through an implicit (only defined by a successor function) language-independent LTS model.

In this paper, we present a way to extend the CADP environment in order to include the C language as an input formalism in this environment, using model extraction and reusing the model-oriented functionality of all the modules in this toolbox. However, due to the complexity of large C programs, we should also take care of the state space explosion problem. To this end, static analysis methods are used to compute program variables that are not needed during the verification of specific properties. Thus, the model can be optimized by eliminating the program variables from the state vector in order to obtain reduced state spaces. In summary, the main contributions of the paper are the following:

1. a model-extraction method to generate both an *abstract control flow graph (ACFG)* and state space as implicit LTSs from concurrent C programs;
2. an influence analysis framework based on model-checking techniques to optimize the state space generation; and
3. the implementation of both proposals as extensions of the standard CADP distribution.

The model-extraction method relies on two main features. On one hand, the operating system APIs, like the socket API, are modeled. This means that any program using the API can have one fixed model of the API calls. On the other hand, an abstract model of the program is built using metamodeling techniques provided by XML. The metamodeling approach facilitates the generation of various output representations, namely implicit LTSs, to perform verification.

Compared with related works, this paper presents several novel contributions. From the point of view of representing C code with high level models, LTSs can be used by the wide range of applications which constitutes CADP, and models are not limited by a single analysis tool, as in the case of SPIN/PROMELA [7]. Moreover, different representations of the same program can be extracted using the LTS model. For instance, our model-extraction approach can generate both an implicit ACFG and an implicit program state space. The method applied in this paper to achieve influence analysis using *on-the-fly* model checking is also new, and it allows us to present new and well-known data flow analyses in a unified manner in terms of a local and incremental resolution of boolean equation systems (BESs). This technology is central to various on-the-fly verification problems on finite-state concurrent systems, like model checking, equivalence checking, partial order reduction and test generation [12]. On-the-fly techniques aim at solving the analysis problem by dynamically constructing only those parts of the model that are necessary for computing the results. Because resolution time and memory complexities of alternation-free BESs are linear in the size of the program model [13], the use of BESs is an appropriate and efficient way to solve data flow analysis. Section 5 contains a more detailed comparison with related work.

The implementation of model extraction and static analysis is done within the OPEN/CÆSAR environment of CADP. This framework presents a modularized architecture that decomposes tools into three main components: graph module, exploration module, and libraries module. This approach eases the integration of new languages and tools to the CADP toolbox. To verify C code with CADP, two new modules have been added to it: C.OPEN and ANNOTATOR. On one hand, C.OPEN uses an interface provided by OPEN/CÆSAR to generate a graph module that describes both the implicit LTSs for ACFG and the state space. On the other hand, ANNOTATOR adds a static analysis module to the generic OPEN/CÆSAR environment, and it makes use of the generic CÆSAR_SOLVE library of CADP that is dedicated to local resolution of alternation-free BESs. Thanks to these two new modules, the different existing CADP tools can now perform bisimulation, reduction, or verification of C concurrent programs based on well-defined APIs.

This paper is organized as follows: in Section 2, we show the basis of our C model extractor in OPEN/CÆSAR; Section 3 describes the fundamentals of static analysis within OPEN/CÆSAR. In Section 4, we present the two new tool components, namely C.OPEN and ANNOTATOR as well as some experimental results and verification scenarios on two real C concurrent programs. The comparison with related work is given in Section 5. Finally, Section 6 concludes and considers future directions of work.

2. OPEN/CÆSAR-compliant model extractor for the C language

Introducing errors in C concurrent programs is usually very easy (like those using shared memory or message passing for the communication between processes). Moreover, the detection of these errors with the traditional method of testing (*i.e.*, the manual execution of the processes trying to detect any error) is a very complex task, and, after many executions,

we cannot ensure that our programs are free of errors. Therefore, formal techniques like model checking are very suitable for verifying the correctness of these programs.

CADP (*Construction and Analysis of Distributed Processes*)[1] [10] is a toolbox for multiple specification languages (LOTOS, MCRL, SDL, networks of communicating automata, etc.), step-by-step simulation, rapid prototyping, verification, test generation, and performance evaluation of concurrent processes with message-passing communication, *i.e.*, *asynchronous* systems. For that purpose, CADP provides an extensive set of efficient analysis tools, such as for model checking, testing, or performance evaluation tools. It has been successfully applied in domains like avionics (Airbus) [14], multiprocessor architectures (*e.g.*, verification of crucial parts of Tera10 [10], France's most powerful supercomputer) and bioinformatics (*e.g.*, verification of genetic, metabolic, and signaling networks [15]). In this section, we provide a method to verify concurrent software using well-defined APIs with CADP following a model-extraction methodology. These APIs provide access to the functionalities offered by the operating system through external system calls. Both the behavior of the external call to the API and the C code using external functions are modeled into a formal description. This formal description is then verified with CADP.

In the context of CADP, the model-extraction technique consists of generating an LTS according to the OPEN/CÆSAR interface. A program model is generated from the C code, and from the definition of each external function call belonging to the well-specified API. Two different implicit LTSs can be generated from the program model: an *abstract control flow graph* (ACFG) for static data analysis and a program state space, both of which are compliant with the CADP toolbox. An ACFG is a specialized control flow graph where vertices are the program points and edges correspond to the information needed for the static analysis, namely the list of program variables used and defined in the program instruction fired between the program points.

2.1. Concurrent program model

A program model, called *process graph*, can be defined as an automaton $\Gamma = (S, s_0, L, T, E)$, where S is the set of states of the process graph and $s_0 \in S$ is the initial state. L is the set of labels that contain blocks of C code or system calls. $T = S \times L \times S$ contains the possible transitions in the graph. As usual, each transition $(s_1, l, s_2) \in T$ represents the execution of the label l that leads the process from state s_1 to s_2. Finally, $E \subseteq S$ is the set of end states.

Fig. 1 shows the algorithm that generates the process graph. Each sentence s in the code is analyzed to establish whether it is an external API call, a control flow statement, or a procedure call. In the last two cases, each sentence in s is also analyzed to determine whether it contains API calls. The operational model semantics defined for each API is used to create a model for each external call. An example of this semantics is given in [16] for a linux device driver. The transformation of each external API call basically consists of adapting its function parameters to the specification of its model: if the model contains additional arguments, like control error variables, they will be added in the model, whereas parameters that are not necessary will be removed. Then, control flow statements and procedure calls have to be unfolded if they contain any API calls. Otherwise, they are simply treated as a common C statement. Thus, the statement `generate s transitions` at line 14 of Fig. 1 studies the type of control flow statement of s (it decides whether s is an `if`, `if .. else`, `switch-case`, `for`, `while` or `do ... while` statement), and it creates the necessary nodes and transitions in the process graph to represent the control flow condition. Moreover, the statement `generate s transitions` executes the main procedure that generates the process graph to unfold the body of the control flow statement in a recursive way. The other C statements that do not contain any external calls are grouped into a single transition in the process graph. These basic blocks are considered to be atomic: no context switches can occur in the middle of these blocks. They only read and modify local variables. Global variables are accessed through asynchronous communication, semaphores, or other synchronization mechanisms, and they can be referenced with the API model for shared memory. Atomic blocks are used to decrease the number of global states generated in the system. Notice that the source code has to be available to analyze it and manage procedure calls. In summary, the set L contains two types of labels: atomic blocks of C statements that do not include any external call and labels that represent a single system call.

The code in Fig. 2 shows an implementation snippet of Peterson's mutual exclusion (PME) algorithm [17]. The PME protocol is a well-known concurrent algorithm that is used to avoid two or more processes from simultaneously accessing common shared data. To prevent data from being in an erroneous and inconsistent state, the PME protocol protects *critical sections* of code that accesses shared data so that other processes that read from or write to the data are excluded from running.

In the case of the PME implementation, for example, some shared memory functions are used. We use the API model given in Table 1. It defines four basic shared memory functions that are used in the PME code, namely `screate`, `swrite`, `sread`, and `sclose`. In the model, the shared memory is organized in regions (denoted as *reg*. in Table 1) which allocate the shared variables. Each region is identified by a region name assigned by the `screate` function which has three arguments: *reg*.*name*, *reg*.*size*, and *data*. If *reg*.*name* does not exist in the shared memory, *screate* adds a new region with the size and default data indicated by the *reg*.*size* and *data*, and it returns the region identifier *reg*.*id* that is associated with this region. Otherwise, if *reg*.*name* was previously added to the shared memory, `screate` just returns the region identifier *reg*.*id* of

[1] `http://www.inrialpes.fr/vasy/cadp`.

```
 0:  Input Array[] of sentences procedure
 1:  Node next,n=s0; Array[] of sentences cblock
 2:  for each sentence s in procedure
 3:    if s is API call
 4:      generate C code transition (n,cblock,next)
 5:      cblock=empty
 6:      n=next
 7:      next=generate API call transition (n,s)
 8:      n=next
 9:    else if s is (control flow sentence or procedure call)
10:      if s contains API calls
11:        next=generate C code transition (n,cblock)
12:        cblock=empty
13:        n=next;
14:        next=generate s transitions (n,s)
15:        n=next
16:      else
17:        add s to cblock
18:      endif
19:    else
20:      add s to cblock
21:    endif
22:  endfor
23:  if cblock not empty
24:    next=generate C code transition (n,cblock)
25:    cblock=empty
26:  endif
27:  end=next
```

Fig. 1. Process graph generation algorithm.

```
     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;
       ...
 8:    ...
 9:    pid = (pid + 1) % 2;
10:    while ((*(int *) sread (flag1_des) == 1)
10':   (*(int *) sread (turn_des) == 1)){
11:      printf ("Waiting for process %d \n", pid);
       }
       /*********************************************/
       /* Critical section */
12:    pid = (pid + 1) % 2;
13:    printf ("Process %d is in critical section \n", pid);
       /* end of critical section*/
14:    flag0_value=0;
15:    ...
```

Fig. 2. Snippet of Peterson's mutual exclusion C code.

that region. In addition, each region of the shared memory internally stores the number of processes accessing it. The other functions use the region identifier *reg*.*id* to access some shared variable. Function `swrite` writes *data_size* bytes of *data* in the region specified by *reg*.*id*, and returns an error code if *reg*.*id* does not exist or if *data_size* is greater than the data size specified during the creation of the region with `screate`. To read the data from the shared memory, `sread` takes *reg*.*id* and returns the stored data or returns *null* if *reg*.*id* is not in the shared memory. Finally, `sclose` decreases the number of

Table 1
Shared memory API model functions.

Func.	screate	swrite	sread	sclose
arg 1	char* reg.name	int reg.id	int reg.id	int reg.id
arg 2	int reg.size	void* data		
arg 3	void* data	int data_size		
return	int reg.id	int code	void* data	int code

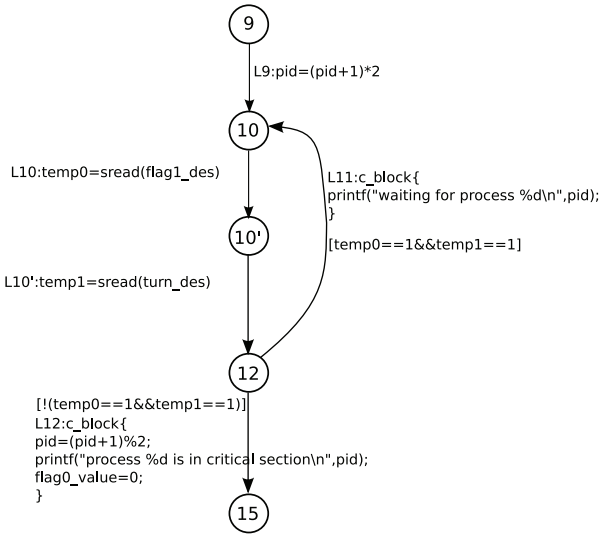

Fig. 3. PME process sub-graph.

references to *reg.id*, and eliminates region *reg.id* if the number of references reaches zero. If *reg.id* does not exist, `sclose` returns an error code. Other software model checker tools such as BLAST, which do not focus on the analysis of well defined APIs, do not employ such elaborate methods for modeling the external call transformation, but they give users the possibility of implementing *dummy* functions with the prototype of the original call.

When applied to the PME program, the process graph in Fig. 3 is obtained where each state corresponds to a program point (numbered in the comments in Fig. 2), and each edge in the graph represents an external function call or an atomic C code block that is labeled using the algorithm in Fig. 1. It is worth noting that the task of constructing the API model is feasible with only moderate effort. In practice, the code that is necessary to model each API call is much smaller than the current implementation in the operating system. Most of the work consists in defining the behavior of a set of related calls in such a way that these calls can be executed in a closed environment, updating the data structures to be used in our model graph. As an example, the entire API model described in Table 1 was implemented in 600 lines of C code, whereas a shared memory library (`shm.[c,h]` and `util.[c,h]`) which is part of IPC in Linux is about 2 350 lines of code. In [16], we checked the behavior equivalence of the implementation of external functions with their true implementation.

2.2. Abstract control flow graph (ACFG)

In the context of live variable analyses, a traditional approach to represent the set of used and defined program variables is the construction of the so-called *define-use chain* [18,1]. This data structure lists all the read accesses that are associated with each write access to a program variable. This structure needs to be constructed *a priori* for the whole program, and it does not allow the application of demand-driven, graph-based analysis methods. In this paper, we are not only interested in live variable analyses but also in data flow analysis based on forward or backward computation and set union or set intersection operators. In order to build a representation of the program behavior that is independent of the program language and type of data flow analysis, and one that allows graph-based analysis algorithms, we have chosen to construct a *labeled transition system* (LTS), whose states correspond to the program points and whose labels only contain the aspects that are relevant to the program property under analysis [19]. We have identified a list of program elements that can be grouped and labeled to form a language-independent description of program instructions. Table 2 provides this list of program elements and their corresponding *abstract* labels. Hence, all variables modified in a program instruction are listed after the keyword `:MODIFY`. The same applies for all variables used in a program instruction; they are listed after the keyword `:USE`, and so forth. It is then possible to construct an LTS from the standard *control flow graph* of the program, where each label corresponds to an abstract program instruction. This LTS is called an *abstract control flow graph* (ACFG) and it is defined

Table 2
Translation of program instructions into abstracted labels.

C program instruction		Label
Type	Example	
Assignment	$y = \ldots$;	:MODIFY y
Load	$\ldots = \ldots y \ldots$;	:USE y
Boolean	$(\ldots > \ldots)$	:BOOL
API call	*read*(…);	:API
Assertion	*assert*(…);	:ASSERT

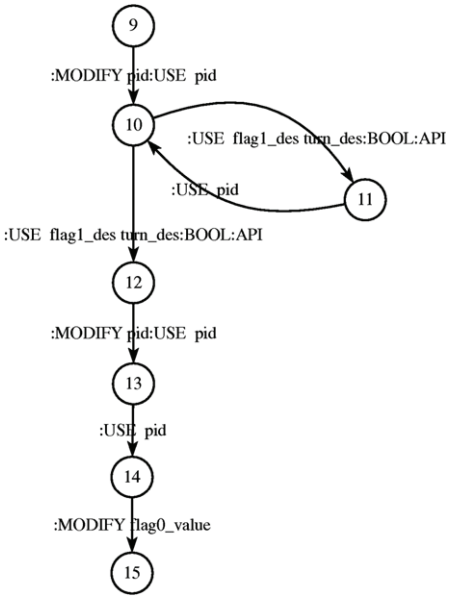

Fig. 4. ACFG of the PME C code snippet prior to influence analysis.

as a tuple $M = \langle S, A, T, s_0 \rangle$, where S represents the set of states of the ACFG (typically the execution points of the program), A represents the abstract labels of the program statements as defined above, $T = \langle s_1, a, s_2 \rangle$ with $s_1, s_2 \in S$ and $a \in A$ represents the set of transitions of the ACFG and s_0 is the initial state. The abstract information in the ACFG labels enables users to verify a specific category of properties about the program. By adding or removing information in the ACFG, different analyses can be achieved. For example, to perform the influence analysis [20] of the PME C code *w.r.t.* the shared memory model presented in Section 2.1, the compiler constructs the implicit ACFG (partly) described in Fig. 4 (the full ACFG contains 20 states and 20 transitions), where the instructions of the PME program are abstracted. For the influence analysis, the only information necessary is the API calls, boolean expressions, and used/modified program variables. Each state of the ACFG corresponds to a point in the program, which is indicated in a comment on the left part of the PME C code description in Fig. 2. Labels show the use and modification of four program variables, namely *pid*, *flag*1*_des*, *turn_des*, and *flag*0*_des*, and indicate boolean expressions and calls to the shared memory API. The resulting ACFG can be used directly as input by a static analyzer computing the influence analysis of the program variables. If we wanted to perform a standard analysis like the *very busy expressions* analysis, the abstract labels would be enhanced with a list of non-trivial arithmetic and boolean expressions used in each program instruction [19].

2.3. Implicit program state space

In CADP, an implicit state space is defined by the implementation of an interface provided by the OPEN/CÆSAR environment. This interface provides the user with the representation of the state and label that are parts of the underlying implicit LTS, as well as the necessary primitives to deal with them. States store information about the system environment. This information, which is known as state vector in model checking, can be global data of the system processes, like channels in sockets or shared memory structures, as well as local data of each process. In order to minimize the size of the state space, some techniques like influence analysis are used to store only the relevant data needed in each state to verify the system. It is worth noting that some information on local and global data is always present in the state vector, namely the number of running processes, their actual state, pid, and type.

In an LTS, a label represents the action performed to evolve from one given state to the following one. We can distinguish between two kinds of actions: the ones executing an external call and the ones executing a block of C statements. The first group is represented by an index that references each different external call and by the value of the parameters used. The second group of labels, resulting from executing an atomic block of C code, is represented by the symbol "i", which indicates an invisible or internal transition. Transitions "i" are local to a process and do not affect external calls.

Moreover, OPEN/CÆSAR provides two special primitives that are related to the transition between states. On one hand, there is a primitive for initializing the system, which generates the initial state, *i.e.*, it creates processes and it initializes the data of local processes. On the other hand, there is another primitive that is responsible for generating all the successors from a given state with its associated label.

Finally, transitions from state to state are constructed in two phases. First, for every system process, all outgoing transitions are obtained from its process graph. Then, the associated code for each transition obtained is executed, resulting in an appropriate update of the state vector variables and the production of the corresponding LTS label.

This model-extraction approach allows us to obtain a complete description of the program state space in an implicit LTS, which is constructed dynamically using the successor function.

3. OPEN/CÆSAR-compliant data flow analyzer for implicit LTSs

Traditional techniques require an *a priori* construction of the whole control flow graph before starting the analysis. In this section, we present *demand-driven*, also called *on-the-fly*, static analysis techniques that dynamically construct the underlying control flow graph during the analysis. We illustrate our approach on an extended version of the *live variables* (*LV*) analysis, called *influence analysis* (*IA*), which has recently been used to reduce the state space of a program via the so-called *abstract matching* technique [21]. We will show how the boolean equation system formalism enables these demand-driven analyses and we will illustrate this formalism on the PME C code example.

3.1. Influence analysis

Let $M = \langle S, A, T, s_0 \rangle$ be an ACFG, as defined in Section 2.2. Observe that, in this graph, states only represent control points that the program flow can visit during execution. Therefore, we can define a data flow analysis over M as a function $dfa : S \to 2^D$ that attaches a set of denotations, elements of D, which give us safe information about the real data that may occur at s during any program execution with each program point $s \in S$. The manner in which D is defined depends on the analysis to be carried out. For instance, in the classic *live variable analysis* ($LV : S \to 2^D$), the goal is to attach each program point with the set of variables that are *alive* there, *i.e.*, those variables whose value at this point may be needed later. Thus, in this case, D must be defined as the set of all program variables, and $x \in LV(s)$ means that variable x is alive at point s.

Influence analysis ($IA : S \to 2^D$) was developed following the same criteria as classic data flow analysis. However, in this case, the goal is to compute all *influential* variables w.r.t. a given set of program sentences that are declared as being *of interest* (*e.g.*, API calls) for each program point. Variable v is *influential* at program point s, if its current value at s may be needed later either directly or indirectly through the value of other variables in some boolean expression or program instruction *of interest*. Basically, *IA* extracts variables in each program point like *LV* analysis does but only considering a subset of interesting instructions or expressions. The objective of this analysis is clear: when a variable does not influence the expression of interest, it can be safely abstracted which, in many cases, leads to important reductions in the state space.

In [22], we developed different influence analyses, considering different expressions or instructions of interest. The most precise analysis, denoted as $\text{IA}_{reachability}$ considers all possible program sentences as interesting. On the other hand, $\text{IA}_{assertion}$ also takes into account the variables in *safety properties* described in the code as assertions, which generates sets of variables bigger than $\text{IA}_{reachability}$. Analysis $\text{IA}_{formula}$ is the least precise analysis, but in contrast, it preserves *liveness properties* by considering all variables appearing in the temporal formulas to be verified as influential variables. Finally IA_{API} is an extension that considers the variables contained in *API calls* as interesting. Hereinafter, we will use the case of IA_{API}, which preserves properties that are related toAPI calls, and we will derive a μ-calculus definition for this analysis from the one that defines analysis *LV*.

The following alternation-free μ-calculus formula [23] defines analysis *LV*:

$$\phi_{LV}(v) = \mu Y.(\langle a \mid used(v, a) \rangle \ \text{true}) \vee (\langle a \mid \neg modified(v, a) \rangle \ Y)$$

where $a \in A$, $v \in D$ (D being the set of program variables), $used(v, a)$ is true iff variable v is used (*i.e.*, read) in instruction a, and $modified(v, a)$ is true iff variable v is modified (*i.e.*, defined/assigned) in instruction a.

Observe that formula $\phi_{LV}(v)$ is recursive. The semantics of $\phi_{LV}(v)$, which is usually denoted as $[[\phi_{LV}(v)]]^e_M$, is given in relation to a labeled transition system M (the abstract control graph mentioned above) and an environment $e : VAR \to 2^S$ that associates formula variables (for example, variable Y used above) with a set of program points. The value of $\phi_{LV}(v)$ is calculated by means of a fixpoint operator. The initial set of states τ_0 is composed of the states from which it is possible to evolve through a transition where v is used. The following iteration produces τ_1, such that $\tau_0 \subseteq \tau_1$, with those states that can evolve towards a state in τ_0, through an instruction where v is not modified, and so on. Clearly, by the Tarski theorem, the calculation of the fixpoint ends, and it gives us the set of program points where variable v is alive. As a consequence, we should apply this formula for each program variable v to obtain the expected output of *LV*.

Formula $\phi_{LV}(v)$ can serve us as a guide to construct a temporal formula ϕ_{API} that defines the IA_{API} analysis given in [20]. As will be shown later, the correct ϕ_{API} definition requires using value-based extension of the alternation-free μ-calculus [24]. This extension enables specifying data variables without propositional variables (*e.g.*, $Y(z)$) and parameterized fixpoints in the temporal formulas. In addition to the $used(v, a)$ and $modified(v, a)$ primitives, we also introduce $bool(a)$ and $api(a)$, which respectively test whether a is a boolean instruction or an API call. The same construction applies to the four *IA* variants presented above. We could define them as μ-calculus formulae in [25].

$$\begin{aligned}\phi_{\text{API}} \;=\; & \mu Y(v : var).(\langle a \mid used(v, a) \wedge (bool(a) \vee api(a))\rangle \,\mathsf{true} \\ & \vee \langle a \mid \neg modified(v, a)\rangle \;\; Y(v)) \\ & \vee \langle a \mid modified(z, a) \wedge used(v, a)\rangle \;\; Y(z)\end{aligned}$$

where $a \in A$, and v, z are program variables. Formula ϕ_{API} may be read as follows. For each program variable $v \in var$, the set of program states where v is an influential variable (w.r.t. API calls) is recursively built as the fixpoint of the sequence of sets of states $\tau_0, \tau_1, \cdots$. The initial set τ_0 is constructed with the program points from which it is possible to evolve through a sentence a, such that a uses variable v, and a is a boolean expression or an API call. Observe that boolean expressions must be taken into account because they define the control flow of programs. Now, the remaining sets are constructed by applying the other two branches of the formula. Thus, on one hand, a state s belongs to τ_i if it is possible to evolve from s to a state in τ_{i-1} by means of an instruction a which does not modify v. On the other hand, variable v influences state s, if it is possible to evolve from s to another state s', where another variable z is of influence, through an instruction a that modifies z and uses v. This last case collects those variables that are indirectly influent. Thus, for instance, if z is an influent variable at point s', and we can evolve from s to s' through instruction $z = v$, then it is evident that the value of v at s is influent and must be stored. As commented above, this indirect relation can be defined in a relatively simple manner thanks to use of the value-based extension of the alternation-free μ-calculus.

As an example, given the snippet of PME ACFG in Fig. 4, we can evaluate the IA_{API} MCL formula above to answer the following question: "Does the program variable *pid* influence state 9 of the PME ACFG?". The boolean answer is given by the computation of the parameterized propositional variable $Y(pid)$, which is evaluated to false since no path exists from state 9 leading to a state where *pid* is used directly or transitively in a boolean expression or an API call. This on-the-fly evaluation of a temporal formula does not require the whole ACFG to be constructed in order to terminate. For instance, states 8 and below (7, 6, etc.) are not explored in order to obtain the false value of $Y(pid)$. This technique is particularly adapted to programs that make use of library or third-party code for which it is not always possible to have the source code.

Nevertheless, in order to verify the above formula for different program variables and program points, it would be necessary to realize each computation independently since standard on-the-fly model checkers do not allow the use of previously computed formulae in the evaluation of additional similar temporal properties. That is why a lower level of representation is needed, such as the *boolean equation systems* (BESs) [26,27]. Instead of a high-level translation between flow equations and temporal formulas like the modal MCL, a low-level connection between a static analysis and a BES formalism, has the following advantages: (i) persistent computation results between subsequent BES resolution calls can be used to obtain an efficient overall resolution, since only one structure, the boolean equation system, is computed for a given analysis; (ii) BESs allow static analyses to be combined with other verification techniques in a highly modular way; and (iii) the resolution of a BES can benefit from the numerous optimizations developed in the literature and efficiently implemented in state-of-the-art libraries [13].

3.2. Boolean equation system

In this section, we reformulate the demand-driven resolution of data flow analyses on ACFGs as the local resolution of alternation-free BESs by generalizing the classical procedures used for the alternation-free fragment of the modal MCL (L_μ) [26,27]. Each data flow analysis is described by one generic alternation-free BES that can be applied to any program represented as an ACFG.

Definition (*A Boolean Equation System (BES)* [26,28])**.** B is a set of blocks of fixpoint equations whose left-hand-sides are boolean variables, and whose right-hand-sides are pure disjunctive or conjunctive formulas. Thus, assuming that $\mathcal{X}$ is a set of boolean variables, a BES is a tuple $B = \langle x, M_1, \ldots, M_n\rangle$ which defines the value of $x \in \mathcal{X}$ by means of equation blocks $M_1, \ldots, M_n$. Each block M_i contains a finite number ($m_i > 0$) of fixpoint boolean equations $M_i = \{x_{ij} \overset{\sigma_i}{=} op_{ij}\boldsymbol{X}_{ij}\}_{j\in[1,m_i]}$. Fixpoint operator $\sigma_i = \mu$ (or ν) is the same for each equation of block M_i and establishes which fixpoint (minimal or maximal) is defined by the block. Each equation in M_i defines the value of a boolean variable x_{ij} ($1 \le j \le m_i$) through the conjunction (or disjunction) of a set of boolean variables. Thus, for each equation, $x_{ij} = op_{ij}\boldsymbol{X}_{ij}$, boolean operator $op_{ij} \in \{\vee, \wedge\}$ is fixed, and the set at the right side $\boldsymbol{X}_{ij} \subseteq \mathcal{X}$ represents the variables whose value is joined or intersected. For example $x \overset{\mu}{=} y \vee w \vee z$ could be a fixpoint equation of any block M_i, if $\sigma_i = \mu$, $x_{ij} = x$, $op_{ij} = \vee$ and $\boldsymbol{X}_{ij} = \{y, w, z\}$.

Boolean constants false and true abbreviate the empty disjunction $\vee\emptyset$ and the empty conjunction $\wedge\emptyset$, respectively. A variable x_{ij} depends upon a variable x_{kl} if $x_{kl} \in \boldsymbol{X}_{ij}$. A block M_i depends upon a block M_k if some variable of M_i depends upon a variable defined in M_k. A BES is *alternation-free* if there are no cyclic dependencies between two blocks. The *local*

(or *on-the-fly*) resolution of an alternation-free BES $B = \langle x, M_1, \ldots, M_n \rangle$ consists of computing the value of x by exploring the right-hand sides of the equations in a demand-driven way, without explicitly constructing the blocks. Several on-the-fly alternation-free BES resolution algorithms with linear time and space complexity are available in the literature [24,28]. To the best of our knowledge, no encodings of data flow analyses in terms of BES resolution have been proposed in the literature.

Translation of IA_{API} analysis in terms of BES. Following the translation from state to Boolean formulas of [13], the encoding of IA_{API} analysis in terms of BES is straightforward given the corresponding MCL formula as expressed above. Given an LTS $M = \langle S, A, T, s_0 \rangle$ that describes the ACFG, the resulting alternation-free BES is as follows:

$$x_{s,v} \overset{\mu}{=} (\underbrace{\bigvee_{s \xrightarrow{a} s',} \text{true}}_{(1)}) \vee (\underbrace{\bigvee_{s \xrightarrow{a} s',} x_{s',z}}_{(2)}) \vee (\underbrace{\bigvee_{s \xrightarrow{a} s',} x_{s',v}}_{(3)})$$

where $s, s' \in S$, $a \in A$, $v, z \in D$, (1) represents $used(v, a) \wedge (bool(a) \vee api(a))$, (2) represents $modified(z, a) \wedge used(v, a)$, and (3) represents $\neg modified(v, a)$.

The least fixpoint operator is stated explicitly by the boolean equation. The forward possibility modality ($\langle\langle \ldots \rangle\rangle$) operator translates into disjunction over all successor states. Boolean expressions over states translate into boolean expressions over actions. The IA_{API} formula is translated into a BES with single μ block and variable parameter v of type D defining a variable $x_{s,v}$ for each state and variable pair $(s,v) \in S \times D$, which expresses that variable v is influential at state s w.r.t. a specific API [25]. This BES can be solved using an optimized resolution algorithm based on a depth-first search for disjunctive equation blocks, such as algorithm *A*4 of [13]. Here, the transformation from BES with parameter v into parameterless BES is direct since the parameter v is part of the boolean variable definition. Hence, at most $|D|$ boolean variables will need to be solved for each state of the ACFG before the analysis is terminated.

As earlier, given the snippet of PME ACFG in Fig. 4, we can evaluate the above IA_{API} BES to answer the following question: "Does the program variable *pid* influence state 9 of the PME ACFG?". The boolean answer is given by the computation of the parameterized boolean variable $x_{9,pid}$, which is defined as follows:

$$\begin{aligned}
x_{9,pid} &\overset{\mu}{=} x_{10,pid} \\
x_{10,pid} &\overset{\mu}{=} x_{11,pid} \vee x_{12,pid} \\
x_{11,pid} &\overset{\mu}{=} x_{10,pid} \\
x_{12,pid} &\overset{\mu}{=} x_{13,pid} \\
x_{13,pid} &\overset{\mu}{=} x_{14,pid} \\
x_{14,pid} &\overset{\mu}{=} x_{15,pid} \\
x_{15,pid} &\overset{\mu}{=} \text{false}
\end{aligned}$$

Variable $x_{15,pid}$ is evaluated to false because state 15 does not have any successor state in the ACFG analyzed (empty disjunction $\vee\emptyset$). However, Fig. 4 is a snippet of the whole ACFG. Indeed, the real state 15 would have a successor state, namely 16, which would be connected in turn to state 17, and so forth, up to state 19, the final state of the ACFG. Hence, the BES construction would eventually also end up in a state without a successor. We considered state 15 as the final state in the above example in order to make the computation shorter. The false value of variable $x_{15,pid}$ can be propagated up to variable $x_{9,pid}$, this means that the answer to the question "Does program variable *pid* influence state 9 of the PME ACFG?" is negative. Solving the BES returns that variable $x_{9,pid}$ is false; hence, the program variable *pid* does not influence state 9 of the ACFG. A simple graph traversal over all states of the ACFG would finally indicate (via an algorithm from [19]) that variable *pid* is not influencing any state of the ACFG. This result could be used in the context of abstract matching to reduce the program state vector by eliminating the variable *pid* from it.

In the algorithm from [19], every transition in the ACFG is traversed exactly once per program variable or expression. Furthermore, BES resolutions are linear in the size of the LTS [13]. Since the constructed BES is unique for all states given a variable or an expression, the resolution of already solved boolean variables is done in constant time. Therefore, each call to the algorithm has a worst-case time complexity $O(|S|+|T|)$, considering that the number of tested variables and expressions is significantly smaller than the number of states and transitions. The same bound applies for memory consumption, since, in the worst case, every state will be stored in a set, taking into account that BES resolution has a linear memory complexity.

Following the same translation approach, we reformulated the demand-driven resolution of *LV*, *VBE*, *AE*, and *RD* analyses as well as the four variants of *IA* analyses on LTSs as the local resolution of alternation-free boolean equation systems (BESs) [19,25,20]. *LV*, *VBE*, *AE* and *RD* are perhaps the most famous examples of flow analyses and are meant to portray backward and forward analyses with least and greatest fixpoint. A novelty of our work is the encoding of forward static analyses (available expressions and reaching definitions) but in terms of forward operators (*i.e.*, successor information) in a BES. These analyses are usually defined in the literature using predecessor information. As a consequence, our encoding enables us to obtain the direct on-the-fly resolution of a large variety of data flow analyses.

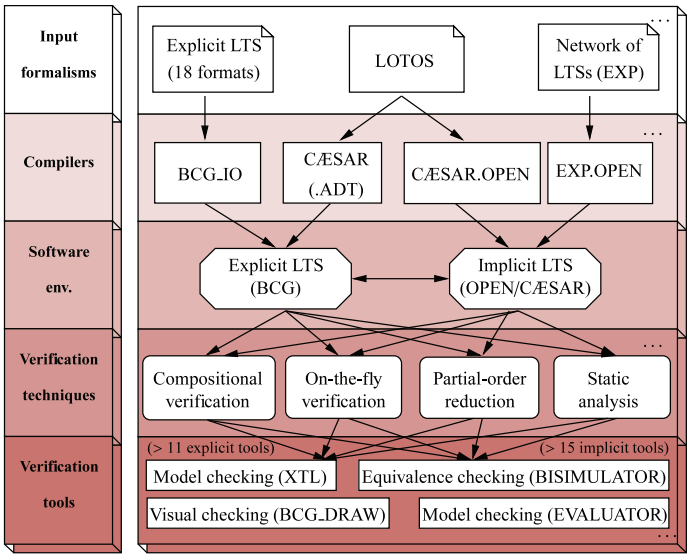

Fig. 5. CADP software environments and functionalities.

4. C.OPEN and ANNOTATOR: CADP components for verifying C programs

CADP is designed in a modular way and puts the emphasis on intermediate formats and programming interfaces, such as the BCG and OPEN/CÆSAR software environments, which allow the CADP tools to be combined with other tools and adapted to various specification languages. Fig. 5 gives an overview of CADP software environments and functionalities.

The toolbox offers compilers for several input formalisms ranging from high-level protocol descriptions written in the ISO language LOTOS to low-level protocol descriptions specified as finite state machines or networks of finite state machines. Numerous forms of verification are supported by CADP: model checking (satisfaction of a modal μ-calculus (MCL) formula by an LTS), equivalence checking (comparison of two LTSs w.r.t. some equivalence/preorder relation), visual checking (graphical inspection of an LTS), performance evaluation (analysis of Interactive Markov Chains represented as LTSs with relevant delays), etc. CADP functionalities are based on several verification techniques, such as compositional, enumerative, on-the-fly, symbolic, and massively parallel verification, as well as partial order reduction and static analysis.

Nevertheless, CADP does not yet offer software model-checking capabilities, such as C or language-independent static analyzers.

Two new tools (named C.OPEN and ANNOTATOR) have been connected to the CADP toolbox [29]. These tools add the possibility to analyze C programs using the CADP environment. While C.OPEN is a C compiler that generates the implicit LTS described in Section 2.1 via the functions given by the CADP graph module interface, ANNOTATOR is a static analyzer that performs on-the-fly data flow analyses (described in Section 3) and program slicing of the ACFGs described in Section 2.2, which are represented as implicit LTSs via the functions of the OPEN/CÆSAR library.

4.1. Software architecture

Fig. 6 shows the link between C.OPEN and ANNOTATOR tools within the CADP toolbox. Both tools can be used jointly or separately.

Jointly. First, C.OPEN extracts the ACFG of the analyzed C concurrent programs and compiles it into the OPEN/CÆSAR intermediate format (*i.e.*, implicit LTS with abstract labels as specified in Section 3). Then, ANNOTATOR statically analyzes the ACFG and determines, for example, the active program variables w.r.t. well-defined API models.

Separately. C.OPEN can directly generate a program state space in the implicit LTS format or use the result returned by ANNOTATOR to construct a reduced one. On the resulting implicit LTS, efficient CADP model checkers can be applied, such as EVALUATOR (evaluation of regular alternation-free μ-calculus formulas) and BISIMULATOR (equivalence checking). Hence, CTL, ACTL, PDL, PDL-Δ, and regular alternation-free μ-calculus properties can be verified on our C input programs, and our C concurrent program can be compared to determine whether it is bisimilar to its specification.

The proposed tools are publicly available through the SMC project web pages.[2] Both new tools are rather small, robust, and mature (in operation for about four years). Detailed manual pages are provided, including more than 25 program examples and step-by-step small case studies.

[2] http://www.gisum.uma.es/tools/smc.

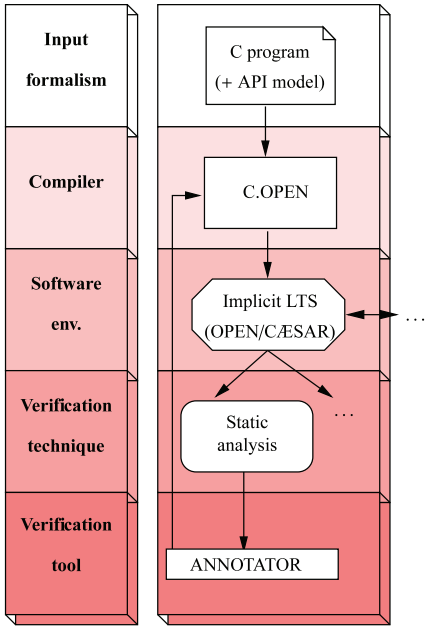

Fig. 6. Extension of CADP (Fig. 5) with C.OPEN and ANNOTATOR tools.

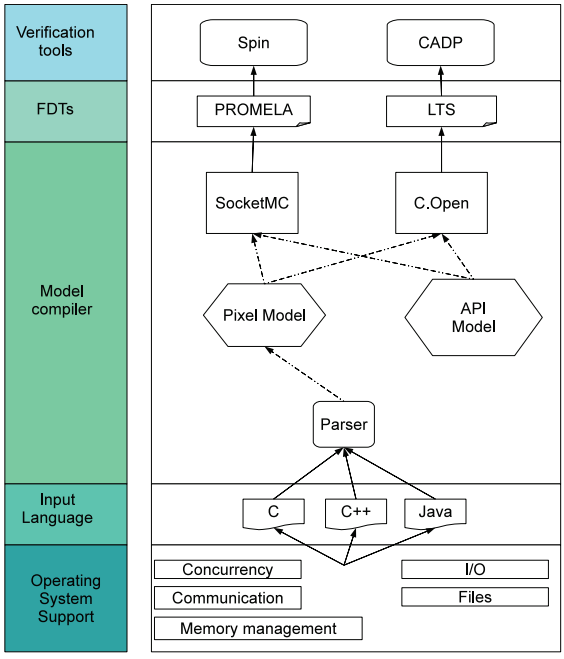

Fig. 7. Model extractor architecture.

4.2. C.OPEN

C.OPEN (Fig. 8) takes a C program as input and connects it to the OPEN/CÆSAR environment. Originally, CADP was designed for the LOTOS and BCG input formalisms. However, many other works have extended the languages and specifications accepted by CADP, namely the FC2 file exchange format, which is an object-oriented language that is called BDL (Behavioural Description Language) [30], timed automata [31], genetic regulatory networks [32], and scheduler specifications based on restrictions [33].

Following the model-extraction approach (Fig. 7), our compiler is composed of two components, called C2XML and XML2LTS, which are written in JAVA and executed in sequence.

Table 3
C.OPEN C language features not supported by C.OPEN.

Feature		Problem	Solution
Functions	Recursive pointer to	Folding not implemented	Not addressed not implemented
Variable modifiers	extern	Global access by all processes	Code transformation: declare this variable as a shared variable
	static	Permanent variable data when exiting function	Code transformation: declare this variable as a global variable in the state vector
	register	Low level access	not addressed

To simplify the generation of the model, the original code is translated into an intermediate specification called PIXL [34]. PIXL is defined as a group of layered XML schemes, each of which contains a set of the most common characteristics of languages that extend communicating finite state machines, such as PROMELA, Statecharts, or SDL. Therefore, PIXL constitutes an intermediate language that represents the model in a consistent way for different analysis tools. C2XML (2000 lines of JAVA code) performs the translation into the PIXL representation of the C program to be analyzed. It is worth noting that C2XML uses JAVACC, which is a parser like lex/yacc in JAVA, with a C grammar to convert C to XML. Then, the XML scheme of the source code generated by C2XML is used as input to XML2LTS(4500 lines of JAVA code) as well as an API model, represented in XML, which indicates how the program model has to be sliced. This application carries out the steps described in Section 2 to generate the OPEN/CÆSAR graph module that describes the implicit LTS, corresponding either to an ACFG or to a program state space. Finally, C.OPEN executes a C compiler (*e.g.*, `cc` or `gcc`) with the resulting graph module and an OPEN/CÆSAR application, like ANNOTATOR or EVALUATOR, to generate an executable CADP verification tool.

4.3. Supported C subset and external functions

C.OPEN can manage most of the common C language statements including loop and selection control flow statements, user functions, castings, arrays, and pointer operators: indirection $*$, reference &, and accessing to members of objects pointed to by variables $\rightarrow$. Moreover, C.OPEN can also deal with pointer arithmetic, one of the less common features of current software model-checking tools for the C language. However, note that some C language features are not supported in C.OPEN like recursive functions or some variable modifiers, like `static` or `register` (see Table 3). One known limitation of the model-extraction approach, which is also observed in the predicate abstraction paradigm used by the BLAST tool, is that it only covers recursive functions if they do not contain well-specified API calls. Indeed, since API function calls are inlined in the model, the inlining process would loop infinitely for recursive calls. Furthermore, recursive API functions would require a dynamic state management to allocate new variables or to implement a more complex heap model. Another C feature that is not supported by C.OPEN is function pointers, this is due to the inlining mechanism for managing functions. Moreover, the current implementation of C.OPEN does not support scope declarations of variables like `static` and `extern` operations that are on register or at a lower level. However this issue can be dealt with using code transformation: static variables can be turned into global variables and the extern ones can be managed as shared variables. As a requisite, C.OPEN takes preprocessed files where all pre-processing instructions have been compiled as input. In our tool, an arbitrary number of variables can be dynamically allocated and heap size is dynamic. However, since the verification framework does not permit the dynamic creation of processes, the number of processes has to be fixed statically.

To analyze programs that include external functions, such as dynamic memory system calls or socket APIs, a model of the external functions that replace the original external functions during the verification must be constructed. As mentioned above, in addition to the behavioral model, it may be necessary to provide an environment abstraction, which models external elements such as buffers, memory heap, or user behavior. We have modeled various commonly used APIs, like the well-known socket API in [35]. This model implements the standard functions to work with sockets: `socket`, `bind`, `connect`, `accept`, `read`, `write`, `close`, and `select`. It provides the abstraction for some operating system facilities, like channels and communication buffers, which are needed to properly model the socket interface. We also provided a dynamic memory model and its heap representation in [16] to verify C programs that use dynamic memory allocation. This model provides functions to allocate and dispose memory (`malloc` and `dispose` functions) and also functions for accessing the memory heap model when using pointer operators over variables pointing to the dynamic memory. In [36], we presented a model of the ARINC 653 interface that models most of the services offered by the interface. Finally, Section 2.1 presents functions for shared memory accesses that create, write, read, and close over shared memory areas for inter-process communication.

4.4. ANNOTATOR

ANNOTATOR (Fig. 8) consists of two parts: a front-end, which is responsible for encoding the static analysis of LTS as a parameterized BES resolution; and a back-end, which is responsible for a parameterized BES resolution, playing the role of the verification engine. The front-end (750 lines of C code on average per analysis) takes as input the LTS that is associated to the ACFG provided by C.OPEN and the type of analysis to be carried out. In the architecture of C.OPEN and ANNOTATOR,

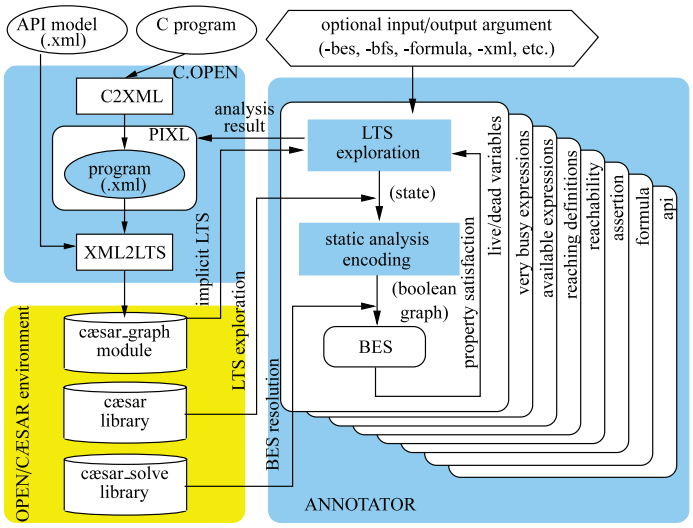

Fig. 8. C.OPEN and ANNOTATOR tool architectures.

the flow from LTS exploration to the static analysis encoding is related to the demand-driven static analysis technique introduced in this paper. Indeed, the implicit LTS description provided by C.OPEN to ANNOTATOR is only a successor function that enables the successor states of a given state to be constructed in the ACFG under analysis. In order for ANNOTATOR to locally build the ACFG, several LTS primitives of the CÆSAR library need to be invoked. We implemented modules that translate each of the four influence analyses and each of the four classical data flow analyses presented in Section 3 into a BES. BESs are represented implicitly by their successor function, in the same way as LTSs in OPEN/CÆSAR. The back-end is obtained by using efficient algorithms of the CÆSAR_SOLVE library [13], which is part of the OPEN/CÆSAR environment in CADP. Globally, the approach to on-the-fly static analysis is to both construct the LTS and the corresponding parameterized BES on-the-fly and to determine the final value of boolean variables of interest. Only the part of both graphs that needs to perform the static analysis is explored incrementally.

As output, ANNOTATOR produces static analysis results such as XML or textual files depending on the option selected by the user. These formats allow post-processing of computed analyses by directly conveying the result as input to compilers reading these formats, such as C.OPEN, which allows further compilation optimizations.

With respect to static analysis functionalities provided by CADP to date, there is only one static analyzer. It is embedded in the CÆSAR compiler [37] and can only treat process algebraic specifications, like LOTOS. ANNOTATOR is the first stand-alone static analyzer connected to the CADP toolbox that is usable at the CÆSAR.OPEN level (the LTS model, which is independent from the input language). The BES transformation mechanism in ANNOTATOR served as the basis for a more advanced static analyzer connected to CADP, namely the DATALOG_SOLVE tool, which analyzes JAVA programs by context-insensitive pointer analysis [38,39].

4.5. Experimental results and verification scenarios in CADP

We performed a series of experiments to investigate the effectiveness of our new C.OPEN and ANNOTATOR tools. Several experiments were devised to compare the results of our data flow analyses in terms of BESs with those observed in the literature [18]. Twelve classical C program examples showing the value of using a data flow analysis to simplify the compilation of a program were analyzed. In addition to the PROMELA examples extracted from the literature [22], ten other C program examples specific to each one of the implemented influence analyses were considered. To show the scalability of our static analyzer, ANNOTATOR was successfully tested on very large ACFGs that were extracted from the VLTS benchmark[3] with up to 10^6 program counters and instructions. These ACFGs were extracted from several programs from the VLTS benchmark, for instance `vasy_1112_5290` and `vasy_2581_11442`, by using the BCG_LABEL tool from CADP. We defined some renaming rules as regular expressions so that BCG_LABEL could rename all labels of these VLTS into labels in the use/def format of ACFGs labels. As a result, the generated ACFGs contain as many program counters and instructions as states and transitions in the VLTS. As an example, the ACFG obtained from the `vasy_1112_5290` benchmark contains 1112 000 program counters and 5290 000 program instructions.

Finally, to show the independence from the language input of our ANNOTATOR tool, a LOTOS description of the Dekker mutual exclusion protocol was also analyzed with the nine currently implemented data flow analyses. All the ACFGs from the C program examples were extracted by the C.OPEN tool.

3 www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.

Table 4
Reachability analysis results on the Peterson algorithm with CADP.

Use of influence analysis	C.OPEN and GENERATOR (CADP)				
	States	Transitions	Time (s)	Memory (MB)	State size (B)
N	719	1312	0.003	20	124
Y	583	1052	0.003	18	120

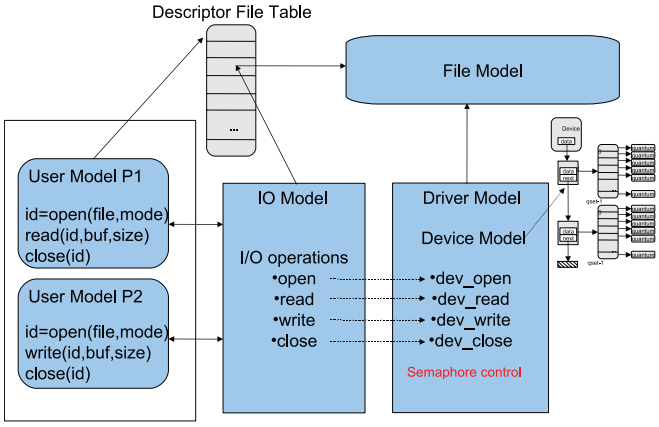

Fig. 9. Component models for the device driver case study.

Our first verification scenario was the evaluation of the three fundamental temporal properties of a mutual exclusion protocol on the C implementation of Peterson's protocol [29]. The PME ACFG was analyzed considering the API influence analysis by the ANNOTATOR tool. As shown in Table 4, C.OPEN used the output result of ANNOTATOR to reduce the PME state space size by 20% (reduction from 719 states and 1 312 transitions to 583 states and 1 052 transitions). We should mention that the IA_{API} static optimization enabled by ANNOTATOR to decrease the program's complexity is not just a removal of dead variables and the code that computes them (which could be computed by a standard live variable analysis). It removes all variables that do not affect the *a posteriori* evaluation correctness of API call related properties. For instance, we successfully checked one safety (mutual exclusion), one liveness (progress requirement), and one fairness (bounded waiting) property on the PME reduced state space with the EVALUATOR model checker of CADP. It is worth noting that the minimized implicit state space resulting from executing C.OPEN and ANNOTATOR can be used as input to other verification tools from CADP (*e.g.*, bisimulation, simulation, testing, etc.).[4]

To the best of our knowledge, well-established model checkers for C, like BLAST and SLAM, cannot be compared with our implemented tools. The analysis of external functions in BLAST requires the user to provide postconditions over the functions or function stubs that limit their behavior. The advantage of this approach is that the analysis is simpler with the corresponding benefits of better execution time and memory consumption. However, this approach does not allow the verification of properties related to the use of external functions, like system calls or functions that depend on the environment. Indeed, BLAST does not provide complex models, like the sockets API or the shared memory API used in the PME example with C.OPEN, to enable the verification of their use. In the case of SLAM, their approach is oriented towards the analysis of drivers for windows systems that use the *Windows Driver Model* (WDM). Our proposal is oriented towards the analysis of more general systems that are not restricted to the use of one particular dynamic memory model like in SLAM. Systems can use other models, like the shared memory model in the PME protocol, which is not supported by SLAM.

Apart from this well-known PME toy example, we considered a second verification scenario inspired by real industrial problems: a device driver for the Linux operating system [40]. The device driver is represented by a driver model and its C implementation, which uses a well-specified dynamic memory API model. In order to verify that the driver satisfies a set of properties, a C program that calls the driver must be designed and verified. Fig. 9 gives the complete model of the case study composed of two user process models (a writer and a reader) and their C implementation as well as a model of the well-specified `stdio` library (IO model) that enables the user processes to access the driver.

This device driver was already analyzed by the SPIN model checker in [16]. From the models defined in this previous work, we adapted the C implementations of the models to the OPEN/CÆSAR environment of CADP. One exception is the dynamic memory API model, which has not been modified. This model can be used directly in our C.OPEN tool. In our approach, the memory model also contains a representation of the heap, whereas in the case of a real implementation, like `malloc`, it would use the heap of the operating system. Our API model is implemented in 1 300 lines of C code whereas `ptmalloc2`, which is the current implementation of `malloc` in the `glibc`, is described in about 5 700 lines of code.

[4] Full implementation, result details, and a thorough discussion on the Peterson case study are available at http://www.gisum.uma.es/tools/smc.

Table 5
Reachability analysis results on the driver model with CADP.

Elements	C.OPEN and GENERATOR (CADP)				
	States	Transitions	Time (s)	Memory (GB)	State size (B)
5	230 132	463 187	4	0.21	806
10	1314 154	2642 134	91	1.3	806
15	3718 241	7245 286	590	3.8	806

Table 6
Reachability analysis results on the driver model with CADP using influence analysis.

Elements	C.OPEN and GENERATOR (CADP)				
	States	Transitions	Time (s)	Memory (GB)	State size (B)
5	213 353	424 037	3	0.19	768
10	1077 164	2146 931	74	1.1	768
15	3416 068	6817 115	550	3.5	768

Table 7
Reachability analysis results on the driver model with SPIN.

Elements	SPIN			
	States	Time (s)	Memory (MB)	State size (B)
10	37 699	0.3	20,7	424
30	359 500	3.15	152.6	424
50	1080 041	11.4	447.5	424
70	2282 522	18.5	939.3	424
100	5171 881	47.3	2100	424

Table 5 gives the results of a reachability analysis of the device driver generated by C.OPEN together with the GENERATOR tool of CADP. The case study is parameterised by the number of elements (from 5 to 15) to write or read through the device driver. Table 7 shows the reachability analysis of the device driver generated by SPIN. Note that state spaces generated with CADP are much larger than the ones generated with the SPIN-based solution. As can be observed in the table, the state space size directly affects the performance results in terms of time and memory. The difference of state space size and performance results can be explained by the optimizations that were manually implemented in the SPIN device drive model. As we want to automatically generate the model in CADP from the C program without any manual implementation, these optimizations were not directly ported to our CADP-based solution. In particular, we think that the number of states of the CADP model can be significantly decreased by allowing blocking guards in statements and by grouping pointer sentences into atomic blocks.

As can be observed, C.OPEN manages device models with up to 15 elements written to or read from the device. It constructs an LTS with more than 3 million states and 7 million transitions on which safety properties, namely deadlock freeness, have been evaluated by the EVALUATOR model checker. C.OPEN was also employed to generate an ACFG for each component `pread.c` (104 states, 119 transitions) and `pwrite.c` (108 states, 124 transitions) of the device driver. Then, ANNOTATOR was used with no modification w.r.t. the version presented in [29] to compute the API influence analysis on each ACFG (resp. in 3 and 4 seconds). ANNOTATOR detected that 9 (resp. 10) variables out of 50 (resp. 51) analyzed variables in `pread.c` (resp. `pwrite.c`) are not necessary to verify temporal formulas on the use of the device driver API. Tables 5 and 6 show that the size of the state spaces can be reduced by a factor of 10% when these influence analysis results are used within C.OPEN.

5. Related work

In the last few years, many tools have been developed to verify C programs. They cover different methods and different features in the input language. Instead of using explicit model checking, SLAM [3,4] and BLAST [5] are software model checkers based on the counterexample-guided abstraction refinement theory. They have been used to check real device drivers with respect to API usage rules. COCCINELLE [6] is a transformation tool for C programs. It identifies API protocols and detects violation of their usage in device-specific code by evaluating CTL properties with standard model checkers. MODEX [7] uses SPIN [41] in the verification of C code. BANDERA [8] verifies JAVA programs by translating the source code into a model that can be expressed in the input language of several model checking tools, namely NUSMV [42] or SPIN. The first version of JPF [9] also models JAVA programs into PROMELA specifications to be verified with SPIN. Other recent projects, like MPI-SPIN [43], are devoted to MPI, modifying the core of the SPIN model checker to analyze parallel programs.

The closest proposal for verifying concurrent C programs based on well-defined APIs is our own work with the tool SPIN. The SOCKETMC [35] tool was developed to verify programs based on the socket interface for process communication. We worked with SPIN further to verify C programs that are connected to the ARINC 653 [36] interface (Avionics Application Software Standard Interface). Another line of work introduced model abstraction in SPIN in order to reduce the complexity of large systems [44]. Moreover, we defined a two-level logic (MALTL) that combined the LTL and CTL logics, as well as LTL and modal mu-calculus, to analyze properties over programs that use dynamic data structures [16,45]. Most of these previous results in model extraction can be reused in the CADP context, thereby allowing the verification of software with the previously defined APIs. One example is the dynamic data model used in Section 4 to verify the device driver with C.OPEN.

With regard to static analysis using *on-the-fly* model checking, the use of BES introduces new contributions compared to other approaches. Traditionally, static analysis is done with global methods, as in standard algorithms based on use-define chains [1,18] and the proposals centered on modal μ-calculus logic (MCL) [23]. In contrast, our approach focuses on demand-driven techniques to solve a considered analysis with no *a priori* computation of the control flow graph. Recently, Zheng and Rugina [46] gave a demand-driven algorithm for an alias analysis which compares favorably to an exhaustive solution, especially in terms of memory consumption. Our technique for data-flow analyses of C programs based on local BES resolution goes in the same direction and provides a novel approach to demand-driven program analyses almost for free, by directly using state-of-the-art BES solvers. To work at the BES level also opens new ways of combining static analysis and model-checking techniques. We could think of combining the BES formulas that represent data flow analyses with BES formulas that represent verification problems similarly to the combination of reduction techniques into equivalence-checking problems in [47]. Moreover, on-the-fly methods are of importance when dealing with realistic, complex programs. Indeed, constructing and handling the program representation becomes a bottleneck for large programs, and dynamic solutions are useful during the design process. In our approach, both LTS and BES are constructed dynamically, thus saving the generation of unnecessary parts of both structures for the given analysis. Another contribution of the paper is the encoding of forward static analyses (*e.g.*, available expressions and reaching definition analyses) only in terms of forward operators (successor transition), whereas such analyses are defined in the literature using predecessor information. The different analyses described are illustrated using examples of C programs and very large state spaces (LTSs) extracted from the VLTS benchmark suite.[5]

6. Conclusion

Models in software are usually used in the specification/design phase and for code generation. Nevertheless, the use of models is also convenient in other phases. This paper presents the analysis of C software with the model-extraction technique. In particular, we use models to support model checking and static analysis. We use two kinds of labeled transition systems to represent both the control flow and the state space of C programs. In addition, we use boolean equation systems as the representation to conduct data flow analysis on LTSs. All these models constitute the core representation to perform efficient verification of C programs with the CADP toolbox.

The experiments carried out using C.OPEN and ANNOTATOR on numerous standard examples assess the functionality of our model-checking framework for C concurrent programs within CADP. This also demonstrates that the modular architecture of our tools allows a rapid integration of new compiler optimizations and new static analyses described as BES resolutions, as well as quick connection to existing compilers and verification tools in CADP. In the PME demonstration, we successfully checked one safety (mutual exclusion), one liveness (progress requirement), and one fairness (bounded waiting) 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. We also verified a realistic driver implementation that manages dynamic memory by constructing an abstracted environment for the standard input/output functions and two processes that access the facilities that the driver provides. However, the driver performance can be improved by adding several optimizations to C.OPEN such as allowing blocking guards in statements or grouping pointer sentences into atomic blocks to decrease the number of states generated.

With respect to future work, our contributions can be extended in several ways. New analyses have been recently designed in the context of program verification, such as the *reset variable* analysis [37]. We could study the adequacy of the BES formalism to model these analyses. Another line of research would be the integration of the new two-level logic (MALTL) [16,45] to CADP in order to analyze temporal properties over programs that use dynamic data structures, as in our Linux device driver example.

References

[1] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2007.
[2] E. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, Cambridge, MA, USA, 2000.

5 www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.

[3] T. Ball, S.K. Rajamani, The slam toolkit, in: G. Berry, H. Comon, A. Finkel (Eds.), Proceedings of the 13th International Conference on Computer Aided Verification CAV'01, Paris, France, in: Lecture Notes in Computer Science, vol. 2102, Springer-Verlag, 2001, pp. 260–264.
[4] T. Ball, E. Bounimova, V. Levin, R. Kumar, J. Lichtenberg, The static driver verifier research platform, in: T. Touili, B. Cook, P. Jackson (Eds.), Proceedings of the 22nd International Conference on Computer Aided Verification CAV'10, Edinburgh, UK, in: Lecture Notes in Computer Science, vol. 6174, Springer, 2010, pp. 119–122.
[5] T.A. Henzinger, R. Jhala, R. Majumdar, The blast software verification system, in: P. Godefroid (Ed.), Proceedings of the 12th International SPIN Workshop on Model Checking of Software SPIN'05, San Francisco, CA, USA, in: Lecture Notes in Computer Science, vol. 3639, Springer-Verlag, 2005, pp. 25–26.
[6] J. Brunel, D. Doligez, R. Hansen, J.L. Lawall, G. Muller, A foundation for flow-based program matching: using temporal logic and model checking, in: Z. Shao, B.C. Pierce (Eds.), Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'09, Savannah, GA, USA, ACM Press, 2009, pp. 114–126.
[7] G.J. Holzmann, M.H. Smith, A practical method for verifying event-driven software, in: B. Boehm (Ed.), Proceedings of the 21st International Conference on Software Engineering ICSE'99, Los Angeles, CA, USA, ACM Press, 1999, pp. 597–607.
[8] J. Hatcliff, M.B. Dwyer, Using the Bandera tool set to model-check properties of concurrent java software, in: K.G. Larsen, M. Nielsen (Eds.), Proceedings of the 12th International Conference on Concurrency Theory CONCUR'01, Aalborg, Denmark, in: Lecture Notes in Computer Science, vol. 2154, Springer-Verlag, 2001, pp. 39–58.
[9] G. Brat, K. Havelund, S. Park, W. Visser, Java pathfinder — a second generation of a java model checker, in: Proceedings of the Post-CAV'00 Workshop on Advances in Verification, 2000.
[10] H. Garavel, R. Mateescu, F. Lang, W. Serwe, CADP 2006: A toolbox for the construction and analysis of distributed processes, in: Proceedings of the 19th International Conference on Computer Aided Verification CAV'07, Berlin, Germany, in: Lecture Notes in Computer Science, vol. 4590, Springer-Verlag, 2007, pp. 158–163.
[11] H. Garavel, Open/cæsar: An open software architecture for verification, simulation, and testing, in: B. Steffen (Ed.), Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98, Lisbon, Portugal, in: Lecture Notes in Computer Science, vol. 1384, Springer-Verlag, Berlin, 1998, pp. 68–84. Full version available as INRIA Research Report RR-3352.
[12] C. Joubert, R. Mateescu, Distributed on-the-fly model checking and test case generation, in: A. Valmari (Ed.), Proceedings of the 13th International SPIN Workshop on Model Checking of Software SPIN'06, Vienna, Austria, in: Lecture Notes in Computer Science, vol. 3925, Springer-Verlag, 2006, pp. 126–145. Full version available as INRIA Research Report RR-5880.
[13] R. Mateescu, Caesar_solve: A generic library for on-the-fly resolution of alternation-free boolean equation systems, International Journal on Software Tools for Technology Transfer 8 (2006) 37–56.
[14] H. Garavel, M. Sighireanu, A proposal for coroutines and suspend/resume in e-lotos, in: ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3).
[15] G. Batt, C. Belta, Model checking genetic regulatory networks using gna and cadp, in: Proceedings of the 11th International SPIN Workshop on Model Checking of Software SPIN2004, Springer, 2004, pp. 158–163.
[16] M.M. Gallardo, P. Merino, D. Sanán, Model checking dynamic memory allocation in operating systems, Journal of Automated Reasoning 42 (2009) 229–264.
[17] M. Raynal, Algorithmique du Parallelisme: le Probleme de l'exclusion Mutuelle, Dunod, Paris, 1984.
[18] F. Nielson, H. Nielson, C. Hankin, Principles of Program Analysis, Springer, 2005.
[19] M.M. Gallardo, C. Joubert, P. Merino, On-the-fly data flow analysis based on verification technology, in: R. Drechsler, S. Glesner, J. Knoop (Eds.), Proceedings of the 6th International Workshop on Compiler Optimization meets Compiler Verification COCV'07, Braga, Portugal, in: Electronic Notes in Theoretical Computer Science, vol. 190, Elsevier, 2007, pp. 33–48.
[20] M.M. Gallardo, C. Joubert, P. Merino, D. Sanán, On-the-fly API influence analysis of software, in: P. Merino, M. Bakkali (Eds.), Proceedings of the 2nd International Conference on Science and Technology JICT'07, Málaga, Spain, Spicum, 2007.
[21] G.J. Holzmann, R. Joshi, Model-driven software verification, in: S. Graf, L. Mounier (Eds.), Proceedings of the 11th International SPIN Workshop on Model Checking of Software SPIN'04, Barcelona, Spain, in: Lecture Notes in Computer Science, vol. 2989, Springer-Verlag, 2004, pp. 76–91.
[22] P. de la Cámara, M.M. Gallardo, P. Merino, Abstract matching for software model checking, in: A. Valmari (Ed.), Proceedings of the 13th International SPIN Workshop on Model Checking of Software SPIN'06, Vienna, Austria, in: Lecture Notes in Computer Science, vol. 3925, Springer-Verlag, 2006, pp. 182–200.
[23] B. Steffen, Data flow analysis as model checking, in: T. Ito, A.R. Meyer (Eds.), Proceedings of the International Conference on Theoretical Aspects of Computer Software TACS'91, Sendai, Japan, in: Lecture Notes in Computer Science, vol. 526, Springer-Verlag, 1991, pp. 346–365.
[24] R. Mateescu, Local model-checking of an alternation-free value-based modal mu-calculus, in: A. Bossi, A. Cortesi, F. Levi (Eds.), Proceedings of the 2nd International Workshop on Verication, Model Checking and Abstract Interpretation VMCAI'98, Pisa, Italy, University Ca' Foscari of Venice.
[25] M.M. Gallardo, C. Joubert, P. Merino, Implementing influence analysis using parameterised boolean equation systems, in: Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation ISOLA'06, Paphos, Cyprus, IEEE Computer Society Press, 2006, pp. 321–329.
[26] H.R. Andersen, Model checking and boolean graphs, Theoretical Computer Science 126 (1994) 3–30.
[27] B. Vergauwen, J. Lewi, A linear algorithm for solving fixed-point equations on transition systems, in: Proceedings of the 17th Colloquium on Trees in Algebra and Programming CAAP '92, Rennes, France, in: Lecture Notes in Computer Science, vol. 581, Springer-Verlag, Berlin, 1992, pp. 322–341.
[28] A. Mader, Verification of Modal Properties Using Boolean Equation Systems, VERSAL 8, Bertz-Verlag, Berlin, 1997.
[29] M. Gallardo, C. Joubert, P. Merino, D. Sanán, C.open and annotator: Tools for on-the-fly model checking c programs, in: D. Bosnacki, S. Edelkamp (Eds.), Proceedings of the 14th International SPIN Workshop on Model Checking of Software SPIN'07, Berlin, Germany, in: Lecture Notes in Computer Science, vol. 4595, Springer-Verlag, 2007, pp. 268–273.
[30] J.-P. Talpin, A. Benveniste, B. Caillaud, C. Jard, Z. Bouziane, H. Canon, BDL, a language of distributed reactive objects, in: K. Kim, K. Mori, E. Nett (Eds.), Proceedings of the 1st International Symposium on Object-Oriented Real-Time Distributed Computing ISORC'98, Kyoto, Japan, IEEE Computer Society, 1998, pp. 196–205.
[31] J. Julliand, H. Mountassir, E. Oudot, VeSTA: A tool to verify the correct integration of a component in a composite timed system, in: Proceedings of the 9th International Conference on Formal Engineering Methods ICFEM'07, Florida, USA, in: Lecture Notes in Computer Science, vol. 4789, Springer, 2007, pp. 116–135.
[32] G. Batt, D. Bergamini, H. de Jong, H. Garavel, R. Mateescu, Model checking genetic regulatory networks using GNA and CADP, in: S. Graf, L. Mounier (Eds.), Proceedings of the 11th International SPIN Workshop on Model Checking of Software SPIN'07, Barcelona, Spain, in: Lecture Notes in Computer Science, vol. 2989, Springer, 2004, pp. 158–163.
[33] J.J.S. Penas, T. Arts, VoDkaV tool: model checking for extracting global scheduler properties from local restrictions, in: J. Lilius, F. Balarin (Eds.), Proceedings of the 3rd International Conference on Application of Concurrency to System Design ACSD'03, Guimaraes, Portugal, IEEE Computer Society, 2003, pp. 247–248.
[34] M.M. Gallardo, J. Martínez, P. Merino, P. Nuñez, E. Pimentel, Pixl: Applying XML standards to support the integration of analysis tools for protocols, Science of Computer Programming 65 (2007) 57–69.
[35] M.M. Gallardo, P. de la Cámara, P. Merino, D. Sanán, Checking the reliability of socket based communication software, International Journal on Software Tools for Technology Transfer 11 (2009) 359–374.
[36] P. de la Cámara, M.M. Gallardo, P. Merino, Model extraction for ARINC 653 based avionics software, in: D. Bosnacki, S. Edelkamp (Eds.), Proceedings of the 14th International SPIN Workshop on Model checking of Software SPIN'07, Barcelona, Spain, in: Lecture Notes in Computer Science, vol. 4595, Springer, 2007, pp. 243–262.

[37] H. Garavel, W. Serwe, State space reduction for processing algebraic specifications, Theoretical Computer Science 351 (2006) 131–145.
[38] M. Alpuente, M. Feliú, C. Joubert, A. Villanueva, Datalog-based program analysis with BES and RWL, in: Proceedings of the 1st International Workshop, Datalog 2.0, Oxford, UK, March 16–19, 2010. Datalog 2.0, in: Lecture Notes in Computer Science, State-of-the-Art Surveys, vol. 6702, Springer, 2011.
[39] M. Alpuente, M. Feliú, C. Joubert, A. Villanueva, Using datalog and boolean equation systems for program analysis, in: D. Cofer, A. Fantechi (Eds.), Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems FMICS'08, L'Aquila, Italy, in: Lecture Notes in Computer Science, vol. 5596, Springer-Verlag, 2009, pp. 215–231.
[40] J. Corbet, A. Rubini, G. Kroah-Hartman, Linux Device Drivers, 3rd ed., O'Reilly Media, Inc., 2005.
[41] G. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (1997) 279–295.
[42] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, Nusmv: a new symbolic model checker, International Journal on Software Tools for Technology Transfer 2 (2000) 410–425.
[43] S.F. Siegel, Verifying parallel programs with MPI-spin, in: F. Cappello, T. Hérault, J. Dongarra (Eds.), Proceedings of the 14th European Parallel Virtual Machine and Message Passing Interface User's Group Meeting PVM/MPI'07, Paris, France, in: Lecture Notes in Computer Science, vol. 4757, Springer, 2007, pp. 13–14.
[44] M.M. Gallardo, J. Martinez, P. Merino, E. Pimentel, αSPIN: A tool for abstraction in model checking, International Journal on Software Tools for Technology Transfer 5 (2004) 165–184.
[45] M.M. Gallardo, D. Sanán, Verification of dynamic data tree with mu-calculus extended with separation, in: Proceedings of the 8th International Conference on Software Engineering and Formal Methods SEFM'10, Pisa, Italy, IEEE Computer Society Press, 2010, pp. 211–221.
[46] X. Zheng, R. Rugina, Demand-driven alias analysis for C, in: Proc. 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages POPL'08, ACM Press, 2008, pp. 197–208.
[47] R. Mateescu, E. Oudot, Bisimulator 2.0: An on-the-fly equivalence checker based on boolean equation systems, in: Proceedings of the 6th ACM & IEEE International Conference on Formal Methods and Models for Co-Design MEMOCODE'08, Anaheim, CA, USA, IEEE Computer Society, 2008, pp. 73–74.