Implementing Influence Analysis using Parameterised Boolean Equation Systems

María del Mar Gallardo, Christophe Joubert and Pedro Merino
University of Málaga
Campus de Teatinos s/n,
29071, Málaga, Spain
{gallardo,joubert,pedro}@lcc.uma.es

***Abstract*— The well-known problem of state space explosion in model checking is even more critical when applying this technique to programming languages, mainly due to the presence of complex data structures. One recent and promising approach to deal with this problem is the construction of an abstract and correct representation of the global program state allowing to match visited states during program model exploration. In particular, one powerful method to implement *abstract matching* is to fill the state vector with a minimal amount of relevant variables for each program point. In this paper, we combine the on-the-fly model checking approach (incremental construction of the program state space) and the static analysis method called influence analysis (extraction of significant variables for each program point) in order to automatically construct an abstract matching function. Firstly, we describe the problem as an alternation-free value-based μ-calculus formula, whose validity can be checked on the program model expressed as a labeled transition system (LTS). Secondly, we translate the analysis into the local resolution of a parameterised boolean equation system (PBES), whose representation enables a more efficient construction of the resulting abstract matching function. Finally, we show how our proposal has been elegantly integrated into CADP, a generic framework for both the design and analysis of distributed systems and the development of verification tools.**

I. INTRODUCTION

One of the most exciting challenges in the model checking community is to apply automatic reachability based verification to standard programming languages. Actually, there are many ongoing projects oriented to adapt the results on formal method research to languages like Java (see BANDERA [16] and JPF [3]) or C/C++ (see VERISOFT [14], FEAVER [18] or SOCKETMC [5]). As expected, a common problem to these approaches is how to deal with the *state space explosion* problem, resulting from the size of data structures employed in real software, which is several orders of magnitude superior to the size of models written with formal description techniques.

Abstract interpretation is one well-established solution to automatically construct smaller and sound models, which may be analyzed by model checking tools (see [7], [16], [10], [2]). This method, employed in tools like JPF, BANDERA or αSPIN, is partial, because it consists in constructing an over-approximation of the program, where non-realistic paths are possible. Here, we are interested in a more recent approach, which tries to solve the problem using precise abstractions. Thanks to a minimal amount of information, such a method explores exactly the paths required for a given property. One technique of particular interest is *abstract matching*. It consists in using a function for reducing the state vector by ignoring variables, whose values are not relevant to check the property. Actually, these variables are temporally replaced by their abstractions, allowing to cut the exploration paths. Moreover, this approach generates an under-approximation of the whole state space. Thus, it never produces non-realistic paths. Holzmann and Joshi were the first in [17] to propose the technique, then employed in [26]. In [4], the authors proposed the use of *static analysis* algorithms to automatically construct abstraction functions. The method makes use of the property to be analyzed, and in practice, it is based on computing the influence graph for each program variable. In this paper, we intend to automatically construct abstract matching functions by performing the influence analysis described in [4] using model checking techniques.

The idea of using model checking to implement static analysis was first expressed by Steffen in [29], who provided a framework to characterize data flow analyses as the verification of particular modal formulas. Schmidt then extended Steffen's work in [27] to relate it with abstract interpretation. More recently, the tool jABC [21] put in practice Steffen's proposals in the context of Java programs. Our approach is close to these previous works, but rather focus on one specific analysis: *influence analysis*. We show how influence analysis can be expressed as an alternation-free modal μ-calculus formula with data parameters evaluated on a labeled transition system (LTS) expressing the abstracted program behavior. Another interesting contribution of the paper is the encoding of influence analysis in terms of on-the-fly Boolean Equation System (BES) resolution. BESs allow a natural description of numerous verification problems, such as model checking of temporal formulas, bisimulation, partial order reduction, horn clause resolution, abstract interpretation and conformance test case generation [19]. Moreover, BESs are efficiently supported by different resolution algorithms in the literature, one implementation being the CÆSAR_SOLVE library [24], which is part of the widespread verification toolbox CADP [13]. This resolution library is used by the model checker EVALUATOR 3.5 [24], but also by bisimulation and partial-order reduction tools. In addition, it has recently been extended with distributed algorithms, thus allowing an immediate distribution of each tool connected to CÆSAR_SOLVE [19]. Hence, our

static analysis proposal can directly benefit from this verification platform. Parallelly, the SOCKETMC tool is now being rewritten [11] for OPEN/CÆSAR, thus creating a complete set of tools to perform the whole cycle towards verification of software with abstract matching functions.

This paper is organized as follows. Section II summarizes the influence analysis algorithms used to construct abstract matching functions. Section III translates the different algorithms into alternation-free μ-calculus formulas with data parameters, and explains the limitations of such an approach. Section IV further transforms the problem into PBES resolutions. Section V shows the implementation and experiment of the different encodings into the verification toolbox CADP. Finally, Section VI gives some concluding remarks and directions for future work.

II. INFLUENCE ANALYSIS FOR ABSTRACT MATCHING

As proposed in [17], an abstract matching function `f()` should be invoked when it is necessary to compact the state vector. In such cases, the abstraction function computes abstract representations of the hidden data and copies the result onto the state vector. In [17], the authors do not address any particular method to generate `f()`, however they present necessary conditions to define sound abstract functions that preserve CTL properties.

In [4] a particular method is proposed to construct `f()` in such a way that the function be sound and oriented to the property to be checked. This method is based on the identification of variables that *influence* the verification result from the current state. In particular, the authors of [4] developed the so-called *influence analysis* (IA) to annotate each program point p with the set of *significant* variables $\mathsf{IA}(p)$ needed to correctly analyze a given property. Data flow analysis IA is a variant of the classic live variable analysis (LV) that attaches each program point with the set of *live* variables at this point. The key difference is that IA makes use of the property to be checked to determine the set of *needed* variables. Informally, a variable is needed (w.r.t. IA), if its current value may be necessary to evaluate the property of interest in the future. Thus, at a given point, a *live* variable (w.r.t. LV) may not be *needed*, if its value does not influence the property evaluation.

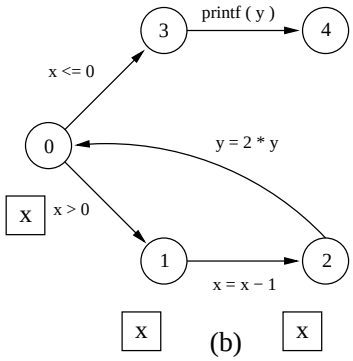
For each program point p, $\mathsf{IA}(p)$ is iteratively calculated as the fixed point of an operator that informally works as follows. Let $\mathcal{V}$ be the set of program variables. IA starts by attaching to p the set $I(p) \subseteq \mathcal{V}$ of variables, which are initially needed at p. The definition of $I(p)$ depends on the property to be analysed. Now, assume that it is known that variable $x \in \mathcal{V}$ is needed at point p, then variable $y \in \mathcal{V}$ *influences* x at p, if there exists an execution path in the program from p to an assignment $x = exp$, and the current value of y is used to calculate exp. The notion of *influence* is recursive since it may be necessary to check if y influences some variable appearing in expression exp in order to decide whether y is needed at point p. As shown in the following sections, a consequence of this recursive behaviour is that we need to use *parameters* when translating IA into μ-calculus formulas or boolean equation systems.

Influence analysis is used in a dual manner by hiding (abstracting) the variables, which *are not* needed at each program point, while the rest of variables remains explicit in the state vector. Therefore, the *best* IA analysis is the one attaching the smallest set of variables to each point.

The work in [4] describes four different influence analyses preserving specific properties. The most precise analysis, denoted as IA_1, only preserves information on reachable code. As an example, we can consider the C process $p1$, shown in Figure 1 (a). The goal of IA_1 is to determine, in each program point (represented as labels $L_0, \cdots, L_4$ in process $p1$, and vertices in the corresponding control flow graph illustrated in Figure 1 (b)), which variables will affect the program execution flow.

```
          void p1 ( int n ) {
               int x = n;
               int y = 1;
/* L0 */       while ( x > 0 ) {
/* L1 */            x = x - 1;
/* L2 */            y = 2 * y;
               };
/* L3 */       printf ( y );
/* L4 */ }
```

(a)



(b)

Fig. 1. Example of a C program $p1$ (a) and its control flow graph (b)

Figure 1 (b) shows the intended result of IA_1 for $p1$. For this process, the static analysis associates the set $\{x\}$ with the labels $L0$, $L1$, and $L2$ (represented in the control flow graph as nodes 0, 1 and 2). Hence, if we are interested in knowing whether a particular label of process $p1$ is reachable, we only have to store variable x at labels $L0$, $L1$, and $L2$. In particular, variable y may be completely hidden because its value is not relevant for this analysis.

The other variants of IA extend IA_1 in the following way: IA_2 produces bigger sets of variables, but it preserves *safety properties*. It extends IA_1 considering variables contained in assertions; IA_3 studies the case of models with global variables; IA_4 is the least precise analysis, but in contrast,

it preserves *liveness properties*. It is based on considering as influencing variables all variables appearing in the temporal formulas to be verified. More details on these influence analyses can be found in [4]. It is worth noting that they can be directly applied to different kinds of modelling and programming languages. In particular, in the rest of the paper, we assume concurrent systems written in C code, although the techniques are applicable to a broader set of imperative languages, such as SystemC, Java, Promela, LOTOS, as far as their control flow graph can be described by our LTS formalism, given in the sequel.

III. INFLUENCE ANALYSIS USING MODEL CHECKING

This section is devoted to one implementation of influence analysis over finite LTSs [4] using model checking. We first define the LTS model extracted from the program being statically analysed, next we describe how the influence analysis problem can be translated into the model checking of temporal formulas over the program model, and finally we give the limitations of such an approach.

A. Presentation of the program model

Influence analysis takes as input a program, or more precisely, a model extracted from it. In this work, we consider the *Labeled Transition System* (LTS) model, which is suitable for value-passing languages, in particular for concurrent system descriptions. An LTS is a tuple $\langle S, A, T, s_0 \rangle$, where:

- S is a finite set of states. Since influence analysis works on program control flow graph, states only contain information about program counter;
- A is a finite set of actions. An action $a \in A$ is represented as a list $i\vec{w}$, where i identifies its type and $\vec{w}$ is a list of typed values;
- $T \subseteq S \times A \times S$ is the transition relation. A transition $(s, a, s') \in T$, also noted $s \xrightarrow{a} s'$, states that the system can move from s to s' by executing action a (s' is an a-successor of s);
- $s_0 \in S$ is the initial state.

In particular, since influence analysis makes use of program variables, that are present in program assignments and in boolean and arithmetical expressions, we use the type Var to denote the set of program variables, and two additional types to denote actions being present in LTS labels:

- $BOOL\ \vec{v}$ describes a boolean expression based on the list of variables $\vec{v}$ of type Var;
- $ASSIGN\ v_1\vec{v}$ describes an assignment expression, where variable v_1 of type Var is assigned a value based on variables $\vec{v}$.

In order to facilitate the further analysis task, we assume that actions in LTSs present a particular form, which depends on the static analysis to be carried out. These abstract actions may be automatically constructed from the instructions on the source language. As for an example, we obtain the LTS on Figure 2 corresponding to the program presented on Figure 1.

Its construction results from the control flow analysis of the program together with a labelling of relevant (i.e., $BOOL$ and $ASSIGN$) and invisible (i.e., τ) actions. Moreover, our model splits each action "$BOOL\ v_1,\ \ldots,\ v_j$" in actions "$BOOL\ v_i$" containing only one variable v_i, for all $i \in [1, j]$. A non-deterministic choice is then made between actions "$BOOL\ v_i$". Similarly, each action "$ASSIGN\ v_1\ v_2\ \ldots\ v_j$" is split in actions "$ASSIGN\ \ v_1\ \ v_i$" with two variable parameters only, for all $i \in [2, j]$. We can also remark that non-determinism may be introduced artificially (i.e., actions "$BOOL\ x$" from state 0) when creating the LTS. However, since the unique purpose of such an LTS is to enable influence analysis, all pertinent information for the analysis is kept.

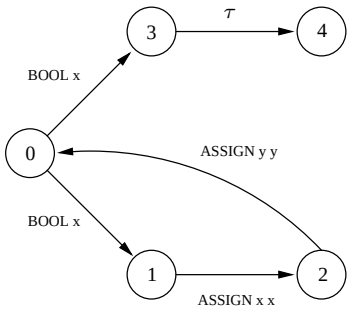


Fig. 2. Example of LTS extended with special actions BOOL and ASSIGN

B. Influence analysis using L^1_μ formulas with data parameters

Modal μ-calculus [20] is an expressive temporal logic based on fixed points, that allows to express a wide range of properties on LTSs, including those of various other useful logics, such as PDL [9] or CTL [6] (as well as its action-based extension ACTL [25]).

The alternation-free fragment of the modal μ-calculus, noted L^1_μ [8], is obtained by forbidding mutual recursive dependencies between minimal and maximal fixed point variables. This logic is of practical usefulness thanks to the existence of linear resolution algorithms in the size of the formula (number of operators) and LTS (number of states and transitions).

In this work, we are interested in the value-based extension of the logic [23], which enables the specification of data variables and parameterised fixed point into the temporal formulas. Properties are not restricted to static label description, but they can refer to dynamic values dependent from the system execution. Formulas of alternation-free value-based modal μ-calculus are defined by the following grammar (where $X \in \mathcal{X}$ is a propositional variable, and $\mathcal{X}$ a set of propositional variables):

$$\begin{aligned} \phi \quad ::= \quad & \mathsf{false} \mid \mathsf{true} \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \langle a \rangle\, \phi \mid [a]\, \phi \\ & \mid X(\vec{e}) \mid \mu X(\vec{x} : \vec{t} := \vec{e}).\phi \mid \nu X(\vec{x} : \vec{t} := \vec{e}).\phi \end{aligned}$$

The semantics of a formula ϕ over an LTS $M = (S, A, T, s_0)$ denotes the set of states satisfying ϕ and it is defined as follows: boolean operators have their usual definition; possibility operator $\langle a \rangle\, \phi$ (resp. necessity operator $[a]\, \phi$) define states from which some (resp. all) transitions labeled by action a lead to states satisfying formula ϕ. We will use overlined actions $\overline{a}$ to represent all actions different from a; propositional

variables X are parameterised by data variables $\vec{e}$; minimal (resp. maximal) fixed point operator $\mu X(\vec{x} : \vec{t} := \vec{e}).\phi$ (resp. $\nu X(\vec{x} : \vec{t} := \vec{e}).\phi$) denotes the least (resp. greatest) solution of the fixed point equation $X(\vec{x} : \vec{t}) = \phi$, parameterised by data variables $\vec{x}$ and argument types $\vec{t}$, evaluated with the arguments $\vec{e}$ and interpreted over 2^S. On-the-fly model checking determines if the initial state s_0 of an LTS satisfies a formula ϕ and belongs to the set of states denoted by ϕ. Although there only exist partial resolution algorithms for value-based L^1_μ, they are practically useful, especially in the context of influence analysis problems.

As a fragment of the data flow analysis problem, influence analysis has shown to be solvable using model checking techniques [28], namely using the modal μ-calculus. However, there doesn't exist to our knowledge a value-based L^1_μ formula encoding the problem of influence analysis. Our approach is the same in spirit to the one of [21], where checking a program property corresponds to writing a new formula, evaluating it on the model and extracting from the set of states satisfying the formula, those defining the different program points. Considering that influence analysis algorithm IA_1 from [4] attaches each program point with the set of variables, whose value is needed to preserve the reachability graph, the resulting value-based L^1_μ formula is:

$$\begin{aligned}\phi_{\mathsf{IA}_1} = \mu Y(v : Var).\quad & (\langle BOOL\ v\rangle \mathsf{true} \\ & \vee\ \langle ASSIGN\ z : Var\ v\rangle Y(z) \\ & \vee\ \langle \overline{ASSIGN\ v\ z : Var}\rangle Y(v))\end{aligned}$$

ϕ_{IA_1} is true if variable v is influent on the initial state of the LTS. Labels "$ASSIGN\ z : Var\ v$" represent graph actions, where the value of an existentially bound variable z is influenced by v in an assignment expression. On the contrary, "$ASSIGN\ v\ z : Var$" stands for actions, where variable v, given as parameter, is modified by an existentially bound variable z.

Even if encoding influence analysis in terms of value-based μ-calculus formulas over LTS is straightforward, it is a new practically useful result for static analysis tools using model checking (e.g., jABC [21]).

Similarly to IA_1, algorithms IA_{2-4} can be encoded as a μ-calculus formula. Since algorithm IA_2 relies on assertions present in the program, it is necessary to extend abstract actions in the LTS with a new type of label:

- $ASSERT\ \vec{v}$ describes an assertion composed of variables $\vec{v}$ of type Var. Similarly to boolean expressions, we will consider that each action "$ASSERT\ v_1,\ \ldots,\ v_j$" is split in actions "$ASSERT\ v_i$" containing only one variable v_i, for all $i \in [1, j]$.

Hence, ϕ_{IA_1} can naturally be extended by taking into account assertion variables and we obtain the formula:

$$\begin{aligned}\phi_{\mathsf{IA}_2} = \mu Y(v : Var).\quad & (\langle BOOL\ v\rangle \mathsf{true} \\ & \vee\ \langle ASSERT\ v\rangle \mathsf{true} \\ & \vee\ \langle ASSIGN\ z : Var\ v\rangle Y(z) \\ & \vee\ \langle \overline{ASSIGN\ v\ z : Var}\rangle Y(v))\end{aligned}$$

Algorithm IA_3 being an extension of IA_1 and IA_2 considering not only local variables but also global variables, the encoding of the problem as a μ-calculus formula is unchanged and does not need an extra definition. However, algorithm IA_4 aims at preserving generic temporal properties, and for this purpose, all variables included in such a property have an influence over the program execution. Since the information contained in temporal properties is external to the program being checked, it will not be accessible in its extracted model, described as LTS. Hence, a variable x is influent modulo IA_4 at a specific program point if it is included in the set of variables used in the temporal properties or if the evaluation of ϕ_{IA_4}, given as follows, on the LTS is true:

$$\begin{aligned}\phi_{\mathsf{IA}_4} = \mu Y(v : Var).\quad & (\langle BOOL\ v\rangle \mathsf{true} \\ & \vee\ \langle ASSIGN\ w_i : Var\ v\rangle \mathsf{true} \\ & \vee\ \langle ASSIGN\ z : Var\ v\rangle Y(z) \\ & \vee\ \langle \overline{ASSIGN\ v\ z : Var}\rangle Y(v))\end{aligned}$$

The formula ϕ_{IA_4} is an extension of ϕ_{IA_1} with as many modal operations $\langle ASSIGN\ w_i : Var\ v\rangle$ as variables w_i present in the external temporal property. Indeed, if a variable v affects the value of w_i in the program, then v is an influent variable itself.

To illustrate the use of model checking μ-calculus formulas for influence analysis, we can show the result of evaluating ϕ_{IA_1} on the LTS given in Figure 2. Checking the validity of ϕ_{IA_1} for variable x on state s_0 will return true, since there exist boolean expressions (e.g., "$BOOL\ x$") involving x reachable from s_0. This process can be iterated through all states figuring in the LTS and all variables of the program (i.e., x and y), allowing the progressive construction of the list of variables influencing each state (see Figure 3). We can remark that only x influences part of the LTS. Hence, variable y can be totally disregarded without involving any skip of reachable states.

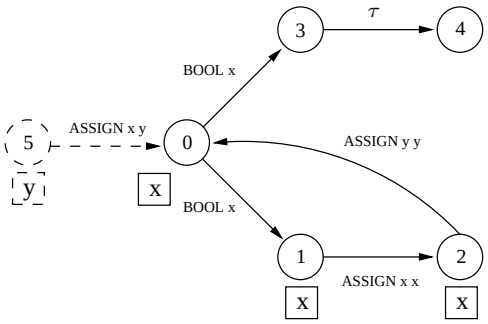


Fig. 3. Example of influence analysis using μ-calculus model checking

C. Limitations of on-the-fly value-based L^1_μ model checking

Instead of iterating through each state, in order to obtain all states satisfying ϕ_{IA_1} for a given variable, it would be more convenient to evaluate only one formula on the whole LTS, and consequently to extract a subgraph from the original LTS, containing all states influenced by the specific variable. This could be done by computing ϕ_{IA_1} on the LTS in a backwards manner using a fixed point iteration. However, this requires the

prior computation of the LTS, and we seek a solution which is suitable for on-the-fly exploration. An adequate μ-calculus formula (for IA_1) would look like the following:

$$\phi_{all\mathsf{IA}_1} = \nu Z. \quad (\ \phi_{\mathsf{IA}_1} \ \wedge \ [\ \mathsf{true} \ ] \ (\ \neg \ \phi_{\mathsf{IA}_1} \vee \ Z \ ) \ )$$

This formula has the same interpretation as ϕ_{IA_1}, meaning that its satisfaction on the initial state s_0 denotes that the given variable is significant for the initial state. Moreover, the on-the-fly evaluation of $\phi_{all\mathsf{IA}_1}$ on a state satisfying ϕ_{IA_1} requires the recursive evaluation of all its successors that also satisfy ϕ_{IA_1}, until all states satisfying ϕ_{IA_1} have been explored. In case of a true answer, it is then possible to draw a positive diagnostic (example), that only reports the states annotated by x in the Figure 3. However, this is only true if x never gets assigned a new value. In such a case, this might create *holes* in the diagnostic, as can be shown in Figure 3 when adding an artificial new state s_5 connected to s_0. Evaluating $\phi_{all\mathsf{IA}_1}$ on s_5 will return false for variable x, whereas x is influent on states s_0, s_1 and s_2. Standard model checkers are not designed to draw such a diagnostic or a partial one with only states satisfying ϕ_{IA_1}. Hence, an iteration through all states is necessary to incrementally construct the set of states influenced by a specific variable.

Working at the level of μ-calculus formulas and standard model checkers allows us to design generic solutions that work not only for influence analysis but, more generally, to many static analyses including classic data flow analyses [21]. However, using on-the-fly model checking is inefficient for influence analysis. Indeed, it would be necessary to use persistent computation results between subsequent costly repetitive evaluations of a same formula over each state of the LTS and each program variable. In this sense, global model checking would be more appropriate, but it is more prone to state space explosion when generating the complete state space and verifying the formula on each of its states. Moreover, it would be more convenient to incrementally generate the list of variables that influence each state, in order to define strategies on which variables need to be checked on successor states, thus allowing a gain in the number of computations needed. To respond to these limitations, a finer-grained encoding of the problem in terms of PBES resolution is preferred and it is described in the following section.

IV. Influence analysis using PBES

This section introduces the Parameterised Boolean Equation System (PBES) model, and gives a PBES encoding of the influence analysis problem.

A. Definition of a parameterised boolean equation system

A *Boolean Equation System* (BES) [1], [22] is a tuple $B = (x, M_1, \ldots, M_n)$, where $x \in \mathcal{X}$ is a boolean variable, $\mathcal{X}$ a set of boolean variables, and M_i are equation blocks ($i \in [1, n]$). Each block $M_i = \{x_{ij} \stackrel{\sigma_i}{=} op_{ij} \boldsymbol{X}_{ij}\}_{j \in [1, m_i]}$ is a set of minimal (resp. maximal) fixed point equations with sign $\sigma_i = \mu$ (resp. $\sigma_i = \nu$). The right-hand side of each equation x_{ij} of block M_i is a pure disjunctive or conjunctive formula obtained by applying a boolean operator $op_{ij} \in \{\vee, \wedge\}$ to a set of variables $\boldsymbol{X}_{ij} \subseteq \mathcal{X}$. 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 block is *closed* if it does not depend upon any other blocks. A BES is *alternation-free* if there are no cyclic dependencies between its blocks. In this case, blocks can be sorted topologically such that a block M_i only depends upon blocks M_k with $k > i$. The *main* variable x must be defined in M_1. In this work, we are interested in the parameterised extension of alternation-free BES [23], called PBES. A PBES is a tuple $B = (x\ (\vec{z} : \vec{t}), M_1, \ldots, M_n)$, where $x \in \mathcal{X}$ is a boolean variable parameterised by data variables in $\vec{z}$ typed by $\vec{t}$. Similarly, each block $M_i = \{x_{ij}(\vec{z_{ij}} : \vec{t_{ij}}) \stackrel{\sigma_i}{=} op_{ij} \boldsymbol{X}_{ij}\}_{i \in [1,n],\ j \in [1, m_i]}$ is parameterised by data variables in $\vec{z_{ij}}$ typed by $\vec{t_{ij}}$.

The *local* (or *on-the-fly*) resolution of an alternation-free PBES $B = (x\ (\vec{z} : \vec{t}), M_1, \ldots, M_n)$ consists in computing the value of $x(\vec{z} : \vec{t})$ by exploring the right-hand sides of the equations in a demand-driven way, without explicitly constructing the blocks. Several on-the-fly BES resolution algorithms [1], [22] and PBES resolution algorithms [23], [15] are available. Although PBES algorithms are partial computations, they are practically useful on finitely branching LTSs.

B. Encoding of influence analysis as PBES resolution

PBES being a relatively new formalism, solving value-based L^1_μ formulas by means of a translation to PBESs isn't as standard as it would be for BESs and L^1_μ formulas.

To solve influence analysis using PBES resolution, the first step is to construct an adequate equation system. Following the approach of [23], it is possible to transform the problem of evaluating a value-based alternation-free μ-calculus formula upon an LTS, into the resolution of a parameterised modal equation system (PMES) upon the LTS, by extracting fixed point operators out of the formula. Starting from ϕ_{IA_1}, the resulting PMES contains one block of modal equations and it is given as follows:

$$\begin{aligned} Y(v : Var) \quad \stackrel{\mu}{=} \quad & (\ \langle BOOL\ v \rangle \ \mathsf{true} \\ & \vee \ \langle ASSIGN\ z : Var\ v \rangle \ Y(z) \\ & \vee \ \langle \overline{ASSIGN\ v\ z : Var} \rangle \ Y(v)) \end{aligned}$$

Then, to obtain a PBES each modal equation block is converted into a boolean equation block by 'projecting' it on each state of the LTS being checked:

$$\begin{aligned} \{Y_s(v : Var) \quad \stackrel{\mu}{=} \quad & \bigvee_{s \xrightarrow{a} s' \ \mid \ a \models BOOL\ v} \mathsf{true} \\ & \vee \bigvee_{s \xrightarrow{a} s' \ \mid \ a \models ASSIGN\ z\ v} Y_{s'}(z) \\ & \vee \bigvee_{s \not\xrightarrow{a} s' \ \mid \ a \models ASSIGN\ v\ z} Y_{s'}(v)\}_{s \in S} \end{aligned}$$

Expressions "$a \models BOOL\ v$", "$a \models ASSIGN\ z\ v$", and "$a \models ASSIGN\ v\ z$" respectively mean that a is a boolean expression containing the program variable v, a is an assignment, where variable v is modified by a variable z,

```
1  INFLUENCE_ANALYSIS (S,A,T,s0) : S → 2^Var
2     visited := {s0}; explored := ∅; var := ∅;
3     while visited ≠ ∅ do
4        s := get(visited); visited := visited \ {s};
5        explored := explored ∪ {s}; d(s) := ∅;
6        forall w ∈ Var do
7           if solve(Y_s(w)) then
8              d(s) := d(s) ∪ {w}
9           endif
10       endfor;
11       forall s → s' | s' ∉ explored do
12          visited := visited ∪ {s'}
13       endfor
14    endwhile;
15    return d
```

Fig. 4. Influence analysis of LTS using PBES resolution

and a is an assignment, where a variable z is modified by variable v.

A boolean variable $Y_s(v)$ is true iff state s satisfies the propositional variable Y considering variable v. Thus, the on-the-fly influence analysis of variable x on the initial state of the LTS amounts to compute the value of variable $Y_{s_0}(x)$.

Generalizing the approach, the influence analysis of all program variables x over all states s in the LTS, can be transformed into an iterative local PBES resolution algorithm.

The function INFLUENCE_ANALYSIS, shown on Figure 4, describes the influence analysis of an LTS $M = \langle S, A, T, s_0 \rangle$ using a PBES resolution for each program variable and LTS state. It starts the resolution with the initial state s_0 (line 2). Given a state s, the function constructs and solves for each program variable w (lines 6–10), the corresponding boolean variable $Y_s(w)$ (line 7, function $solve$). If the variable w is influent upon the current state s, then the set $d(s)$ of influent variables on state s is increased with variable w (line 8). Then, the algorithm iterates through each of the successor states of s (lines 11–13) updating the set of visited states. The analysis continues until all states are explored (line 3). The result of function INFLUENCE_ANALYSIS is the function $d : S \rightarrow 2^{Var}$, which returns for each state, the list of variables that are significant. Such a function d can be further used to automatically construct an abstract matching function stating which variables need to be inserted in the state vector at each program point. Finally, we can also remark that the algorithm presented on Figure 4 can be applied with all influence analysis algorithms IA_{1-4} by using the corresponding PBES encodings when constructing boolean variable $Y_s(w)$ (line 7).

This solution is similar in spirit to the model checking specification in terms of μ-calculus formulas, as it allows to directly provides the desired property as an equation system, whereas it was expressed as a temporal formula in the previous approach. An important aspect of the method is that influence analysis will require the resolution of only one structure, the parameterised boolean equation system, whereas it needed the resolution of as many μ-calculus formulas as variables being checked, multiplied by the number of states in the LTS. Moreover, the PBES is solved on-the-fly, which means that only the relevant parts of it are computed for each state and each variable. Finally, since a boolean variable x_{ij} defined in M_i may be required several times during the resolution process, it is possible to obtain an efficient overall resolution by using persistent computation results between subsequent resolution calls.

V. IMPLEMENTATION AND EXPERIMENTS

Similarly to model checker EVALUATOR 3.5 [24], ANNOTATOR (1 600 lines of C code, see Figure 5) has been fully developed within CADP [13] by using the generic OPEN/CÆSAR environment [12] for on-the-fly exploration of LTSs.

EVALUATOR (*resp.* ANNOTATOR) consists of two parts: a front-end, responsible for encoding the verification of the L^1_μ formula (*resp.* the static analysis type) on LTS_1 as a BES (*resp.* PBES) resolution. EVALUATOR produces also a counterexample by interpreting the diagnostic provided by the BES resolution; and a back-end, responsible of BES (*resp.* PBES) resolution, playing the role of verification engine. Both tools are obtained by using, as back-end, algorithms of the CÆSAR_SOLVE library [24]. Globally, the approach to on-the-fly model checking (*resp.* static analysis) is both to construct on-the-fly the LTS_1 and corresponding BES (*resp.* PBES) and to determine the final value of the main variable.

In the sequel, we present an experimentation with EVALUATOR 3.5 of the influence analysis property IA_1 expressed as a modal equation system (MES) that is not parameterised, and the implementation and experiment of ANNOTATOR to achieve the static analysis of an LTS using PBES resolution within CADP.

A. Experiments with EVALUATOR 3.5

The current EVALUATOR model checker of CADP, whose version is 3.5, does not handle data parameters in μ-calculus formulas. However it is possible to use EVALUATOR 3.5 with the μ-calculus formula ϕ_{IA_1}, by transforming it in a parameterless equation system. This can be done, assuming that the set of program variables x_i is known, by instantiating each call to $Y(x_i)$ into a parameterless propositional variable named Y_{x_i}. Moreover, to get a more compact representation of the expanded formula, we can use modal equation systems (MES), which are accepted as input for EVALUATOR 3.5 as *.blk* files (option *-block*). Such transformation has already been realized in Section IV-B where the formula ϕ_{IA_1} was expanded into a PMES. In order to obtain a resolution complexity linear in the size of the LTS and PMES, it is necessary to simplify the PMES, by splitting each right-hand side equation in order to have a single boolean or modal operator [23]. Simplifying the PMES Y of Section IV-B leads to the following PMES:

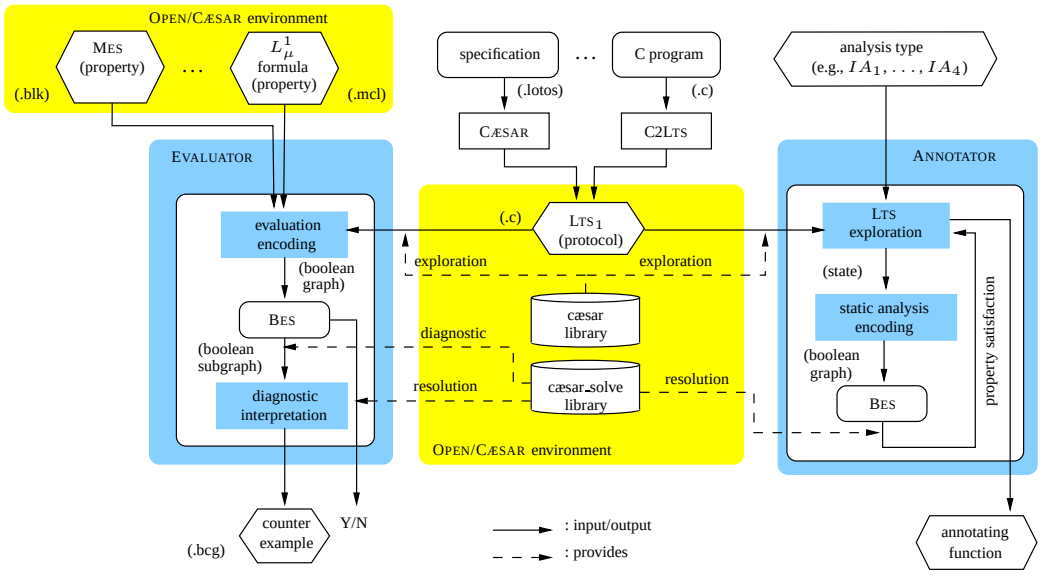


Fig. 5. The on-the-fly tools EVALUATOR and ANNOTATOR

$$
\begin{array}{lcl}
Y_1(v_1 : Var) & \stackrel{\mu}{=} & Y_2(v_1) \vee Y_3(v_1) \\
Y_2(v_2 : Var) & \stackrel{\mu}{=} & \langle BOOL\ v_2 \rangle \ \mathsf{true} \\
Y_3(v_3 : Var) & \stackrel{\mu}{=} & Y_4(v_3) \vee Y_5(v_3) \\
Y_4(v_4 : Var) & \stackrel{\mu}{=} & \langle ASSIGN\ z : Var\ v_4 \rangle\ Y(z) \\
Y_5(v_5 : Var) & \stackrel{\mu}{=} & \langle \overline{ASSIGN\ v_5\ z : Var} \rangle\ Y(v_5)
\end{array}
$$

Next, we transform the simplified PMES in a MES using the parameterless propositional variable Yj_v_i. This MES has a size quadratic w.r.t. the number of influencing variables in the program, but this may be of reasonable size if the number of variables in the program is also not very large. The *.blk* file, for variables x and y in the LTS on Figure 2, is the following:

```
block mu B is
 Y1_x = Y2_x or Y3_x
 Y2_x = < "BOOL x" > TRUE
 Y3_x = Y4_x or Y5_x
 Y4_x = < "ASSIGN y x" > Y1_y
 Y5_x = < not ("ASSIGN x y") > Y1_x
 Y1_y = Y2_y or Y3_y
 Y2_y = < "BOOL y" > TRUE
 Y3_y = Y4_y or Y5_y
 Y4_y = < "ASSIGN x y" > Y1_x
 Y5_y = < not ("ASSIGN y x") > Y1_y
end block
```

Then, to evaluate the influence of variable x (*resp.* y) on the initial state s_0, we can use the *.blk* clause `eval B:Y1_x` (*resp.* `eval B:Y1_y`), which tells EVALUATOR 3.5 which propositional variable it has to check. As a consequence, another limit of the method using EVALUATOR 3.5 is that we cannot check the influence property on a state different from the initial state, as EVALUATOR 3.5 will systematically evaluate the MES on the initial state of the considered LTS.

B. Implementation of an on-the-fly static analyser in CADP

Instead of using current model checkers, we seek a solution that will explicitly manipulate the encoded problem as PBES, implementing the algorithm given in Figure 4. This led us to the need of constructing a static analyser in CADP, based on the OPEN/CÆSAR interface for on-the-fly exploration of LTS. Adapting existing tools to perform influence checking on-the-fly is of real interest.

The architecture of such a tool, named ANNOTATOR, is described on Figure 5. For each visited state in the LTS, it computes the encoding of the static analysis problem in terms of PBES and solves it upon the state following the algorithm in Figure 4. In the case of influence analysis, the corresponding PBES, given in Section IV-B, is projected to the LTS to generate a *flat* (i.e., parameterless) BES, that is solved by the CÆSAR_SOLVE library. Once the satisfiability of the static property has been computed, the tool updates the definition of a function that returns for each state the result of the analysis (i.e., a set of significant variables in the context of influence analysis). After exploring the entire state space, the annotating function results are either returned by the tool or given as input to other applications, e.g., for abstract matching.

Another important feature of the tool is that both the extracted model (as LTS) and the PBES can be constructed and explored on-the-fly, thus allowing incremental exploration of only the part of both graphs that is necessary to perform the static analysis.

ANNOTATOR is fully functional and has been tested on more than 10 experiments, some of them being extracted from the literature [4]. The majority of the tested examples were explicitly defined as *binary coded graphs* (BCG) [13] describing program control flow graphs as LTSs. A first evaluation of the concept showed the elegant and adaptable structure of the tool for different influence analyses, and its

capability to easily integrate new static analyses in terms of BESs. All analyses on tested experiments were immediately solved, principally due to a lack of industrial scale case studies. Once the tool discussed in [11] will be fully implemented, experiments would be far more appealing and we will be able to show the real contribution of influence analysis in tackling the state space explosion problem. Furthermore, there does not exist to our knowledge a benchmark of industrial case studies that would enable comparisons with other tools (e.g., jABC [21]) and demonstrate the added value of using PBESs for influence analysis. Hence, constructing such environment of experimentation is a line of research of first importance.

VI. CONCLUSION AND FUTURE WORK

Static analysis is a necessary step towards software model checking with abstract matching. Our encodings of the influence analysis problem in terms of alternation-free μ-calculus formulas with data parameters and in terms of PBES resolution enables to automatize the analysis process and to use it in conjunction with on-the-fly verification tools. To develop robust explicit-state analysis tools, it is necessary to use efficient and generic verification components. Our on-the-fly static analyser ANNOTATOR goes towards this objective by relying on the generic OPEN/CÆSAR environment [12] for on-the-fly LTS exploration within CADP [13] and by using the BES resolution library CÆSAR_SOLVE [24].

We plan to continue our work along several directions. First, we will show the impact of automatic abstract matching on the explored state space size during verification of C programs thanks to the translator C2LTS proposed in [11]. Next, we will study the interconnection of ANNOTATOR and C2LTS tools integrated into CADP with tools extending SPIN, such as SOCKETMC and αSPIN [10]. Finally, we will seek solutions to other static analysis problems, especially data flow analyses already expressed as μ-calculus formulas in [27], by investigating their translation in terms of BESs resolution.

VII. ACKNOWLEDGEMENTS.

This work has been supported by the Spanish MEC under grants TIN2004-7943-C04 and TIN2005-09405-C02-01. The second author is also supported by a Lavoisier grant from the French Ministry of Foreign Affairs. We are indebted to Radu Mateescu for its valuable feedback on interacting our proposal with CADP model checkers.

REFERENCES

[1] H. R. Andersen. Model checking and boolean graphs. *TCS*, 126(1):3–30, 1994.

[2] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian abstractions for model checking C programs. In *Proc. of TACAS'01*, LNCS vol. 2031, pp. 268–283.

[3] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - A second generation of a Java model checker. In *Proc. of AV'00*.

[4] P. Cámara, M.M. Gallardo, and P. Merino. Abstract Matching for Software Model Checking. In *Proc. of SPIN'06*, LNCS vol. 3925, pp. 182–200.

[5] P. Cámara, M.M. Gallardo, P. Merino, and D. Sanán. Model checking software with well-defined APIs: the socket case. In *Proc. of FMICS'05*, pp. 17–26.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Trans. on Prog. Lang. and Sys.*, 8(2):244–263, 1986.

[7] D. Dams. Abstraction in Software Model Checking: Principles and Practice (Tutorial Overview and Bibliography). In *Proc. of SPIN'02*, LNCS vol. 2318, pp. 14–21.

[8] E. A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proc. of LICS'86*, pp. 267–278.

[9] M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *J. of Comp. and Sys. Sci.*, 18(2):194–211, 1979.

[10] M.M. Gallardo, J. Martinez, P. Merino, and E. Pimentel. αSPIN: A Tool for Abstraction in Model Checking. *Springer Int. J. on Soft. Tools for Tech. Trans. (STTT)*, 5(2-3):165–184, 2004.

[11] M.M. Gallardo, P. Merino, and D. Sanán. Towards Model Checking C Code with OPEN/CÆSAR. In *Proc. of MSVVEIS'06*, pp. 198–201.

[12] H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proc. of TACAS'98*, LNCS vol. 1384, pp. 68–84.

[13] H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *Europ. Assoc. for Soft. Sci. and Tech. (EASST)*, 4:13–24, 2002.

[14] P. Godefroid. Software Model Checking: The VeriSoft Approach. *J. of Formal Meth. in Sys. Design (FMSD)*, 26(2):77–101, 2005.

[15] J.F. Groote and T.A.C. Willemse. Parameterised boolean equation systems. *Th. Comp. Sci.*, 343(3):332–369, 2005.

[16] J. Hatcliff, M. Dwyer, C. Pasareanu, and Robby. Foundations of the Bandera Abstraction Tools. In *The Essence of Comp.*, LNCS vol. 2566, pp. 172–203, 2003.

[17] G.J. Holzmann and R. Joshi. Model-Driven Software Verification. In *Proc. of SPIN'04*, LNCS vol. 2989, pp. 76–91.

[18] G.J. Holzmann and M.H. Smith. Software Model Checking: Extracting verification models from source code. *Soft. Test. Verif. and Relia.*, 11:65–79, 2001.

[19] C. Joubert and R. Mateescu. Distributed On-the-Fly Model-Checking and Test Case Generation. In *Proc. of SPIN'06*, LNCS vol. 3925, pp. 126–145.

[20] D. Kozen. Results on the Propositional μ-calculus. *Th. Co. Sci.*, 27:333–354, 1983.

[21] A-L. Lamprecht, T. Margaria, and B. Steffen. Data-Flow Analysis as Model Checking Within the jABC. In *Proc. of CC'06*, LNCS vol. 3923, pp. 101–104.

[22] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. VERSAL 8, Bertz Verlag, Berlin, 1997.

[23] R. Mateescu. Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In *Proc. of VMCAI'98*, University Ca' Foscari of Venice, 1998.

[24] R. Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *Springer Int. J. on Soft. Tools for Tech. Trans. (STTT)*, 8(1):37–56, 2006.

[25] R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems*. In *Sem. of Sys. of Concur. Proc.*, LNCS vol. 469, pp. 407–419, 1990.

[26] C.S. Pasareanu, R. Pelánek, and W.. Visser. Concrete Model Checking with Abstract Matching and Refinement. In *Proc of CAV'05*, LNCS vol. 3576, pp. 52–66.

[27] D.A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proc. of POPL'98*, pp. 38–48.

[28] D. Schmidt and B. Steffen. Program Analysis as Model Checking of Abstract Interpretations. In *Proc. of SAS'98*, LNCS vol. 1503, pp. 351–380.

[29] B. Steffen. Data Flow Analysis as Model Checking. In *Proc. of TACS'91*, LNCS vol. 526, pp. 346–365.