

# Static 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  $\mu$ -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 may be elegantly integrated into CADP, a generic framework for both the design and analysis of distributed systems and the development of verification tools.

## 1 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 [2]) or C/C++ (see VERISOFT [14], FEAVAR [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,

---

\* This work has been supported by the Spanish MEC under grant TIN2004-7943-C04. The second author is also supported by a Lavoisier grant of the French Ministry of Foreign Affairs

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 [7,16,12,3]). This method, employed in tools like JPF, BANDERA or  $\alpha$ SPIN, 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] and [5]. One novel contribution in [5] is 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 [5] 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  $\mu$ -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 Boolean Equation Systems (BES). BES 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, BES 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 [11]. 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. Parallely, the SOCKETMC tool is now being rewritten for OPEN/CÆSAR (the new tool being called C2LTS), 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 2 summarizes the influence analysis algorithms used to construct abstract matching functions. Section 3 translates the different algorithms into alternation-free  $\mu$ -calculus formulas with data parameters, and explains the limitations of such an approach. Section 4 further transforms the problem into PBES resolutions. Section 5 shows how to experiment the different encodings into the verification toolbox CADP. Finally, Section 6 gives some concluding remarks and directions for future work.

## 2 Influence analysis for abstract matching

As proposed in [17], an abstract matching function  $\mathbf{f}()$  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  $\mathbf{f}()$ , however they present necessary conditions to define sound abstract functions that preserve CTL properties.

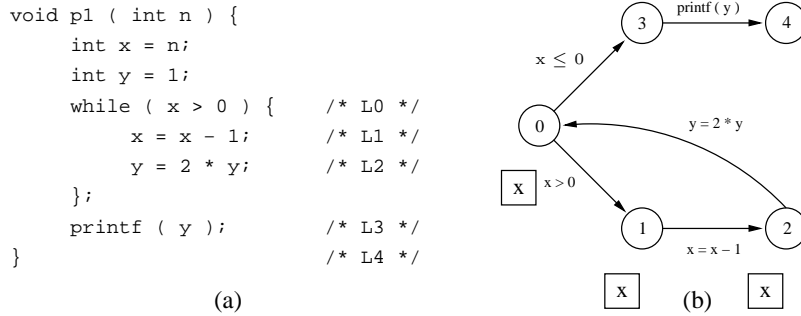
In [5] a particular method is proposed to construct  $\mathbf{f}()$  in such a way that the function be sound and oriented to the property to be checked. This method is based on the identification of variables that *influence* the verification result from the current state. In particular, the authors of [5] developed the so-called *influence analysis* (IA) to annotate each program point  $p$  with the set of *significant* variables  $\text{IA}(p)$  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 evaluation of the property.

For each program point  $p$ ,  $\text{IA}(p)$  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  $x = \text{exp}$ , and the current value of  $y$  is used to calculate  $\text{exp}$ . The notion of *influence* is recursive since it may be necessary to check if  $y$  influences some variable appearing in expression  $\text{exp}$  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  $\mu$ -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 [5] describes four different influence analyses preserving specific properties. The most precise analysis, denoted as  $\text{IA}_1$ , only preserves information on reachable code. As an example, we can consider the C process  $p1$ , shown in Figure 1 (a). The goal of  $\text{IA}_1$  is to determine, in each program point (represented as labels  $L_0, \dots, L_4$  in process  $p1$ , and vertices in the corresponding control flow graph illustrated in Figure 1 (b)), which variables will affect the program execution flow.



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

Figure 1 (b) shows the intended result of  $\text{IA}_1$  for  $p1$ . For this process, the static analysis associates the set  $\{x\}$  with the labels  $L_0$ ,  $L_1$ , and  $L_2$  (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  $L_0$ ,  $L_1$ , and  $L_2$ . In particular, variable  $y$  may be completely hidden because its value is not relevant for this analysis.

The other variants of IA extend  $\text{IA}_1$  in the following way:  $\text{IA}_2$  produces bigger sets of variables, but it preserves *safety properties*. It extends  $\text{IA}_1$  considering variables contained in assertions;  $\text{IA}_3$  studies the case of models with global variables;  $\text{IA}_4$  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 [5]. 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.

### 3 Mu-calculus model checking for influence analysis

This section is devoted to the model-checking of influence analysis over finite LTSS. 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.

### 3.1 Presentation of the program model

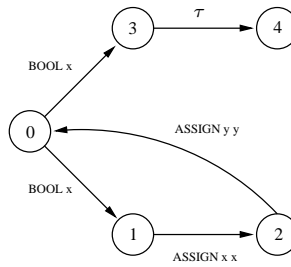
Influence analysis takes as input a program, or more precisely, a model extracted from it. In this work, we consider the *Labeled Transition System* (LTS) model, which is suitable for value-passing languages, in particular for concurrent system descriptions. An LTS is a tuple  $\langle S, A, T, s_0 \rangle$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

- $S$  is a finite set of states;
- $A$  is a finite set of actions. An action  $a \in A$  is represented as a list  $i\mathbf{w}$ , where  $i$  identifies the type of actions and  $\mathbf{w}$  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.

Furthermore, with respect to the influence analysis problem, we are mainly interested in the set of program variables, that are present in program expressions, such as boolean and assignment expressions. Thus, we will use only one type of value, for instance the type  $Var$  denoting the set of program variables, and we define two types of actions being present in LTS labels:

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

*Example 1.* Using the research work of [6,13], which focuses on extracting LTSS out of C programs using well-specified APIs, we can construct an LTS (see Figure 2) corresponding to the program presented on Figure 1.



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

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.,  $\tau$ ) actions. Moreover, our model splits each action “*BOOL*  $v_1, \dots, v_j$ ” in actions “*BOOL*  $v_i$ ” containing only one variable  $v_i$ , for all  $i \in [1, j]$ . Similarly, each action “*ASSIGN*  $v_1 v_2 \dots v_j$ ” 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. Consequently, the static analysis will still have a unique solution.

### 3.2 Influence analysis using $L_\mu^1$ formulas with data parameters

Modal  $\mu$ -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 [4] (as well as its action-based extension ACTL [25]).

The alternation-free fragment of the modal  $\mu$ -calculus, noted  $L_\mu^1$  [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  $\mu$ -calculus are defined by the following grammar (where  $X \in \mathcal{X}$  is a propositional variable, and  $\mathcal{X}$  a set of propositional variables):

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

The semantics of a formula  $\phi$  over an LTS  $M = (S, A, T, s_0)$  denotes the set of states satisfying  $\phi$  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  $\phi$ ; propositional variables  $X$  are parameterised by data variables  $\mathbf{e}$ ; minimal (resp. maximal) fixed point operator  $\mu X(\mathbf{x} : \mathbf{t} := \mathbf{e}).\phi$  (resp.  $\nu X(\mathbf{x} : \mathbf{t} := \mathbf{e}).\phi$ ) denotes the least (resp. greatest) solution of the fixed point equation  $X(\mathbf{x} : \mathbf{t}) = \phi$ , parameterised by data variables  $\mathbf{x}$  and argument types  $\mathbf{t}$ , evaluated with the arguments  $\mathbf{e}$  and interpreted over  $2^S$ . On-the-fly model checking determines if the initial state  $s_0$  of an LTS satisfies a formula  $\phi$  and belongs to the set of states denoted by  $\phi$ .

Influence analysis is a static program analysis process that it is intended to extract from a specification the set of variables influent on the property evaluation for each program control point. Although it is a fragment of the data flow

analysis problem, which has been shown to be solvable using model checking techniques [28], namely using the modal  $\mu$ -calculus, there doesn't exist to our knowledge a value-based  $L_\mu^1$  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  $\text{IA}_1$  from [5] attaches each program point with the set of variables, whose value is needed to preserve the reachability graph, the resulting value-based  $L_\mu^1$  formula is:

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

Similarly, algorithms  $\text{IA}_{2-4}$  can be encoded as a  $\mu$ -calculus formula. Since algorithm  $\text{IA}_2$  relies on assertions present in the program, it is necessary to extend our LTS with a new type of label:

- *ASSERT*  $v$  describes an assertion composed of variables  $v$  of type  $\text{Var}$ .

$\phi_{\text{IA}_1}$  can naturally be extended by taking into account assertion variables and we obtain the following formula:

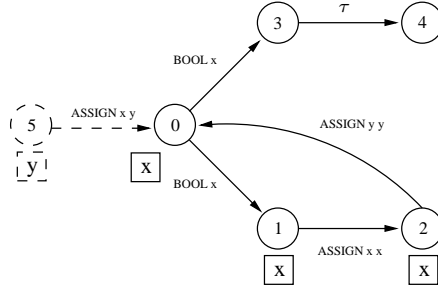
$$\begin{aligned} \phi_{\text{IA}_2} = \mu Y(v : \text{Var} := x). (& \langle \text{BOOL } v \rangle \text{ true} \\ & \vee \langle \text{ASSERT } v \rangle \text{ true} \\ & \vee \langle \text{ASSIGN } z : \text{Var } v \rangle Y(z) \\ & \vee \langle \neg(\text{ASSIGN } v z : \text{Var}) \rangle Y(v))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

Algorithm  $\text{IA}_3$  being an extension of  $\text{IA}_1$  and  $\text{IA}_2$  considering not only local variables but also global variables, the encoding of the problem as a  $\mu$ -calculus formula is unchanged and does not need an extra definition. However, algorithm  $\text{IA}_4$  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, checking the influence  $\text{IA}_4$  of a variable  $x$  at a specific program point is equivalent to, first, test the inclusion of  $x$  in the set of variables used in the temporal properties, then, if  $x$  is not included, evaluate  $\phi_{\text{IA}_4}$  on the LTS as follows:

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

The formula  $\phi_{\text{IA}_4}$  is an extension of  $\phi_{\text{IA}_1}$  with as many modal operations  $\langle \text{ASSIGN } w_i : \text{Var } v \rangle$  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.

*Example 2.* To illustrate the use of model checking  $\mu$ -calculus formulas for influence analysis, we can show the result of evaluating  $\phi_{IA_1}$  on the LTS given in Example 1. Checking the validity of  $\phi_{IA_1}$  for variable  $x$  on state  $s_0$  will return true, since there exists 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  $\mu$ -calculus model checking

### 3.3 Limitations of using on-the-fly value-based $L_\mu^1$ model checking

Instead of iterating through each state, in order to obtain all states satisfying  $\phi_{IA_1}$  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  $\phi_{IA_1}$  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  $\mu$ -calculus formula (for  $IA_1$ ) would look like the following:

$$\phi_{allIA_1} = \nu Z. ( \phi_{IA_1} \wedge [ \text{true} ] ( \neg \phi_{IA_1} \vee Z ) )$$

This formula has the same interpretation as  $\phi_{IA_1}$ , 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  $\phi_{allIA_1}$  on a state satisfying  $\phi_{IA_1}$  requires the recursive evaluation of all its successors that also satisfy  $\phi_{IA_1}$ , until all states satisfying  $\phi_{IA_1}$  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  $\phi_{allA_1}$  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  $\phi_{IA_1}$ . 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  $\mu$ -calculus formulas and standard model checkers, allows to design generic solutions that work not only for influence analysis but, more generally, to many static analyses including data flow analyses [21]. However, using on-the-fly model checking presents limitations such as the reusability of formulas validity for different states given a variable, in order to use previous computations to faster the check of new explored states and variables. In this sense, global model checking would be more appropriate, but 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.

## 4 Influence analysis using PBES

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

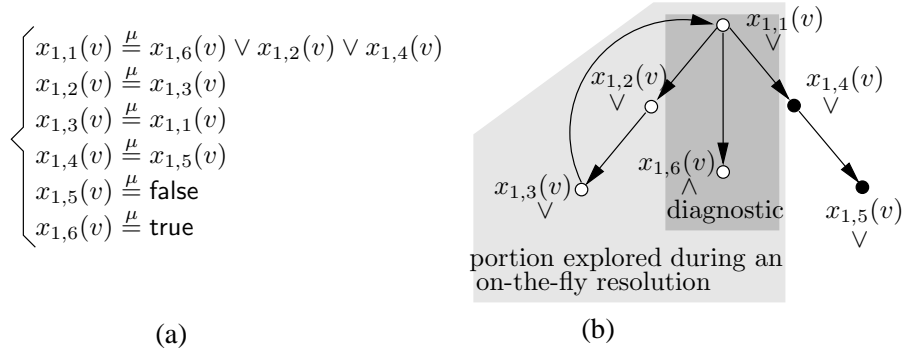
### 4.1 Definition of a parameterised boolean equation system

A *Boolean Equation System* (BES) [1,22] is a tuple  $B = (x, M_1, \dots, M_n)$ , 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  $M_i = \{x_{ij} \stackrel{\sigma_i}{=} op_{ij} \mathbf{X}_{ij}\}_{j \in [1, m_i]}$  is a set of minimal (resp. maximal) fixed point equations with sign  $\sigma_i = \mu$  (resp.  $\sigma_i = \nu$ ). 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  $x_{kl} \in \mathbf{X}_{ij}$ . 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  $B = (x (\mathbf{z} : \mathbf{t}), M_1, \dots, M_n)$ , where  $x \in \mathcal{X}$  is a boolean variable parameterised by data variables in  $\mathbf{z}$  typed by  $\mathbf{t}$ . Similarly, each block  $M_i = \{x_{ij}(\mathbf{z}_{ij} : \mathbf{t}_{ij}) \stackrel{\sigma_i}{=} op_{ij} \mathbf{X}_{ij}\}_{i \in [1, n], j \in [1, m_i]}$  is parameterised by data variables in  $\mathbf{z}_{ij}$  typed by  $\mathbf{t}_{ij}$ .

The semantics  $\llbracket op\{x_1(\mathbf{z}_1 : \mathbf{t}_1), \dots, x_k(\mathbf{z}_k : \mathbf{t}_k)\} \rrbracket \delta$  of a formula  $op\{x_1(\mathbf{z}_1 : \mathbf{t}_1), \dots, x_k(\mathbf{z}_k : \mathbf{t}_k)\}$  w.r.t.  $\mathbb{B} = \{\text{false}, \text{true}\}$  and a context  $\delta : \mathcal{X} \rightarrow \mathbb{B}$ , which must initialize all variables  $x_1, \dots, x_k$ , is the boolean value  $\delta(x_1(\mathbf{z}_1 : \mathbf{t}_1)) \text{ op } \dots \text{ op } \delta(x_k(\mathbf{z}_k : \mathbf{t}_k))$ . The semantics  $\llbracket M_i \rrbracket \delta$  of a block  $M_i$  w.r.t. a context  $\delta$  is the  $\sigma_i$ -fixed point of a vectorial functional  $\Phi_{i\delta} : \mathbb{B}^{m_i} \rightarrow \mathbb{B}^{m_i}$  defined as  $\Phi_{i\delta}(b_1, \dots, b_{m_i}) = (\llbracket op_{ij} \mathbf{X}_{ij} \rrbracket (\delta \odot [b_1/x_{i1}, \dots, b_{m_i}/x_{im_i}]))_{j \in [1, m_i]}$ , where  $\delta \odot [b_1/x_{i1}, \dots, b_{m_i}/x_{im_i}]$  denotes a context identical to  $\delta$  except for variables  $x_{i1}, \dots, x_{im_i}$ , which are assigned values  $b_1, \dots, b_{m_i}$ , respectively. The semantics of an alternation-free PBES is the value of its main variable  $x(\mathbf{z} : \mathbf{t})$  given by the solution of  $M_1$ , i.e.,  $\delta_1(x(\mathbf{z} : \mathbf{t}))$ , where the contexts  $\delta_i$  are calculated as follows:  $\delta_n = \llbracket M_n \rrbracket []$  (empty context because  $M_n$  is closed),  $\delta_i = (\llbracket M_i \rrbracket \delta_{i+1}) \odot \delta_{i+1}$  for  $i \in [1, n-1]$  (interpretation of  $M_i$  in the context of all blocks  $M_k$  with  $k > i$ ).

The *local* (or *on-the-fly*) resolution of an alternation-free PBES  $B = (x(\mathbf{z} : \mathbf{t}), M_1, \dots, M_n)$  consists in computing the value of  $x(\mathbf{z} : \mathbf{t})$  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; here we consider both the approach in [23], giving an algorithm to solve alternation-free PBES, and the approach of [1], formulating the BES resolution problem in terms of a *boolean graph* representing the dependencies between boolean variables.

A boolean graph is a triple  $G = (V, E, L)$ , where  $V = \{x_{ij}(\mathbf{z}_{ij} : \mathbf{t}_{ij}) \mid i \in [1, n] \wedge j \in [1, m_i]\}$  is the set of *vertices* (boolean variables with data parameters),  $E : V \rightarrow 2^V$ ,  $E = \{x_{ij}(\mathbf{z}_{ij} : \mathbf{t}_{ij}) \rightarrow x_{kl}(\mathbf{z}_{kl} : \mathbf{t}_{kl}) \mid x_{kl} \in \mathbf{X}_{ij}\}$  is the set of *edges* (dependencies between variables), and  $L : V \rightarrow \{\vee, \wedge\}$ ,  $L(x_{ij}(\mathbf{z}_{ij} : \mathbf{t}_{ij})) = op_{ij}$  is the *vertex labeling* (disjunctive or conjunctive). An example of PBES with one block ( $i = n = 1$ ) and its associated boolean graph is shown on Figure 4.



**Fig. 4.** (a) Example of a parameterised boolean equation system, (b) its boolean graph and the result of an on-the-fly resolution for  $x_{1,1}(v)$ . Black and white vertices denote false and true variables, respectively.

The resolution of variable  $x(\mathbf{z} : \mathbf{t})$  is performed by a joint forward exploration of the dependencies going out of  $x(\mathbf{z} : \mathbf{t})$  with a backward propagation of stable variables (whose final value is determined) along dependencies; the resolution terminates either when  $x(\mathbf{z} : \mathbf{t})$  becomes stable (after propagation of some stable successors) or when the portion of boolean graph reachable from  $x(\mathbf{z} : \mathbf{t})$  is completely explored. The truth value of  $x(\mathbf{z} : \mathbf{t})$  can be accompanied by a diagnostic, which provides the minimal amount of information needed for understanding its computed value, as shown in the dark grey area on Figure 4.

## 4.2 Encoding of influence analysis as PBES resolution

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  $\mu$ -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  $\phi_{IA_1}$ , the resulting PMES contains one block of modal equations and it is given as follows:

$$Y(v : Var) \stackrel{\mu}{=} ( \langle \text{BOOL } v \rangle \text{ true} \\ \vee \langle \text{ASSIGN } z : Var \ v \rangle Y(z) \\ \vee \langle \neg(\text{ASSIGN } v \ z : Var) \rangle Y(v) )$$

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:

$$\{Y_s(v : Var) \stackrel{\mu}{=} \bigvee_{s \xrightarrow{a} s' \mid a \models \text{BOOL } v} \text{true} \\ \bigvee_{s \xrightarrow{a} s' \mid a \models \text{ASSIGN } z \ v} Y_{s'}(z) \\ \bigvee_{s \xrightarrow{a} s' \mid a \models \text{ASSIGN } v \ z} Y_{s'}(v) \}_{s \in S}$$

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  $Y_{s_0}(x)$ . The resolution of variable  $Y_{s_0}(x)$  on the LTS given in Figure 2 is illustrated on Figure 4, where variable  $x_{1,1}(v)$  corresponds to variable  $Y_{s_0}(x)$ , and variables  $x_{1,j}(v)$  are successors reachable from  $Y_{s_0}(x)$ , w.r.t. the PBES given above. As shown by the white color, meaning a **true** value, of node  $x_{1,1}(v)$ , variable  $x$  is influential on state  $s_0$ . A diagnostic can further be constructed to justify this result by showing a boolean subgraph (in the dark grey area on Figure 4) containing the variables making  $x_{1,1}(v)$  **true**. For instance, it shows variable  $x_{1,2}(v)$ , which is a (“*BOOL*  $x$ ”)-successor of  $x_{1,1}(v)$ , such a transition being the minimal condition for  $x$  to be an influence variable.

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

The function `INFLUENCE_ANALYSIS`, shown on Figure 5, describes the influence analysis of an LTS  $M = (S, A, T, s_0)$  using a PBES resolution for each

```

1  INFLUENCE_ANALYSIS ( $S, A, T, s_0$ )  $\longrightarrow S \rightarrow 2^{v(A)}$  :
2   $visited := s_0; explored := \emptyset;$ 
3  while  $visited \neq \emptyset$  do
4     $s := get(visited); visited := visited \setminus s;$ 
5     $explored := explored \cup s;$ 
6    forall  $v \in var(A)$  do
7      if  $solve(Y_s(v))$  then
8         $d(s) := d(s) \cup v$ 
9      endif
10   endfor;
11   forall  $s' \in succ(s) \setminus explored$  do
12      $visited := visited \cup s'$ 
13   endfor
14 endwhile;
15 return  $d$ 

```

**Fig. 5.** Influence analysis of LTS using PBES resolution

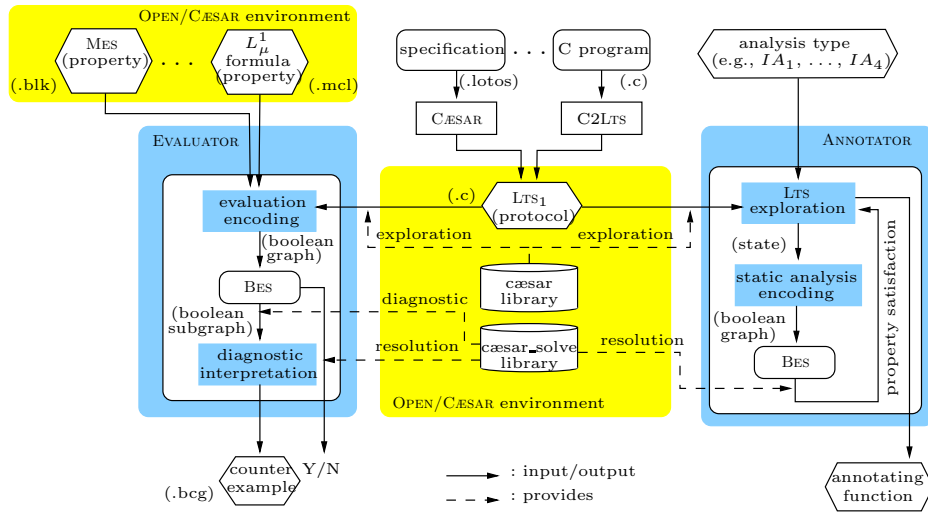
program variable (i.e.,  $v(A)$ ) and LTS state. It starts the resolution with initial state  $s_0$  (line 2) and iterates through each program variable  $v$  (lines 6–10) by constructing and solving the corresponding boolean variable  $Y_{s_0}(v)$  (line 7). If the variable  $v$  is influential upon the current state, then the set  $d(s_0)$  of influence variables for state  $s_0$  is increased with variable  $v$  (line 8). Next, the process constructs the list of successor states of  $s_0$  (lines 11–13), and continues the analysis until all states are explored (line 3). The result of function `INFLUENCE_ANALYSIS` is the function  $d : S \rightarrow 2^{v(A)}$ , 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 5 can be applied with all influence analysis algorithms  $IA_{1-4}$  by using the corresponding PBES encodings when constructing boolean variable  $Y_s(v)$  (line 7).

This solution is similar in spirit to the model checking specification in terms of  $\mu$ -calculus formulas, as it allows to directly provide 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  $\mu$ -calculus formulas as variables being checked, times 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.

## 5 Implementation and experiments

The model checker EVALUATOR 3.5 [24] (see Figure 6) has been developed within CADP [11] by using the generic OPEN/CÆSAR environment [10] for on-the-fly exploration of LTSS. The static analyser ANNOTATOR on Figure 6 is a proposal of tool integrated to CADP, that applies our PBES approach and follows the same architecture of EVALUATOR 3.5.



**Fig. 6.** The on-the-fly tools EVALUATOR and ANNOTATOR

EVALUATOR (*resp.* ANNOTATOR) consists of two parts: a front-end, responsible for encoding the verification of the  $L_\mu^1$  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 structure of ANNOTATOR to achieve the static analysis of an LTS using PBES resolution within CADP.

## 5.1 Experiments with EVALUATOR 3.5

The current EVALUATOR model checker of CADP, whose version is 3.5, does not handle data parameters in  $\mu$ -calculus formulas. However it is possible to use EVALUATOR 3.5 with the  $\mu$ -calculus formula  $\phi_{IA_1}$ , 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 4.2 where the formula  $\phi_{IA_1}$  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]. Thus, simplifying the PMES  $Y$  of Section 4.2 leads to the following PMES:

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

Next, we transform the simplified PMES in a MES using the parameterless propositional variable  $Y_{j-v_i}$ . 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.

## 5.2 Implementation of an on-the-fly static analyser in CADP

Instead of using a model checker, we seek a solution that will explicitly manipulate the encoded problem as PBES, implementing the algorithm given in Figure 5.

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.

The architecture of such a tool, named ANNOTATOR, is described on Figure 6. 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 5. In the case of influence analysis, the corresponding PBES, given in Section 4.2, can be projected to the LTS to generate a *flat* (i.e., parameterless) BES, that would be solved by the CÆSAR\_SOLVE library. Once the satisfiability of the static property has been computed, the tool can update 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 is returned by the tool, and can be further used by 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.

## 6 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  $\mu$ -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 proposition of on-the-fly static analyser ANNOTATOR goes towards this objective by relying on the generic OPEN/CÆSAR environment [10] for on-the-fly LTS exploration within CADP [11] and by using the BES resolution library CÆSAR\_SOLVE [24].

We plan to continue our work along several directions. First, we will finish the construction of ANNOTATOR, as well as the translator C2LTS proposed in [13] and show the impact of automatic abstract matching on the explored state space size during verification. Next, we will study the interconnection of both tools integrated into CADP with tools extending SPIN, such as SOCKETMC and  $\alpha$ SPIN [12]. Finally, we will seek solutions to other static analysis problems, especially data flow analyses already expressed as  $\mu$ -calculus formulas in [27], by investigating their translation in terms of PBES resolution.

*Acknowledgements.* We are indebted to Radu Mateescu for its valuable feedback on the possible interaction of our proposal with CADP model checkers.

## References

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

2. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - A second generation of a Java model checker. In *Proc. of AV'00*.
3. 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.
4. 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.
5. 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.
6. 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.
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. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proc. of TACAS'98*, LNCS vol. 1384, pp. 68–84.
11. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *Europ. Assoc. for Soft. Sci. and Tech. (EASST) Newsletter*, 4:13–24, 2002.
12. M.M. Gallardo, J. Martinez, P. Merino, and E. Pimentel.  $\alpha$ SPIN: A Tool for Abstraction in Model Checking. *Springer Int. J. on Soft. Tools for Tech. Trans. (STTT)*, 5(2-3):165–184, 2004.
13. M.M. Gallardo, P. Merino, and D. Sanán. Towards Model Checking C Code with OPEN/CÆSAR. In *Proc. of MSVVEIS'06*, pp. 198–201.
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  $\mu$ -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.