Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 190 (2007) 33–48
www.elsevier.com/locate/entcs

On-the-Fly Data Flow Analysis Based on Verification Technology ⋆

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

University of Málaga / Gisum
Campus de Teatinos s/n
29071, Málaga, Spain
Email: {gallardo,joubert,pedro}@lcc.uma.es

Abstract

The combination of static and dynamic software analysis, such as data flow analysis (Dfa) and model checking, provides benefits for both disciplines. On the one hand, the information extracted by Dfas about program data may be utilized by model checkers to optimize the state space representation. On the other hand, the expressiveness of logic formulas allows us to consider model checkers as generic data flow analyzers. Following this second approach, we propose in this paper an algorithm to calculate Dfas using on-the-fly resolution of boolean equation systems (Bess). The overall framework includes the abstraction of the input program into an implicit labeled transition system (Lts), independent of the program specification language. Moreover, using Bess as an intermediate representation allowed us to reformulate classical Dfas encountered in the literature, which were previously encoded in terms of μ-calculus formulas with forward and backward modalities. Our work was implemented and integrated into the widespread verification platform Cadp, and experimented on real examples.

Keywords: data flow analysis, model checking, labeled transition system, boolean equation system

1 Introduction

The last two decades have been the most productive from the point of view of techniques and tools for testing and ensuring the reliability of complex software. The investment in formal methods has produced a number of powerful languages, algorithms, methodologies and tools to be successfully employed with software *models*. Today, we can think of applying the same methods to *real programs* in a transparent way, integrating the verification functionality within the compiler.

⋆ This work was supported by Spanish MEC under grant TIN2004-7943-C04. The second author was also supported by a Lavoisier grant from the French Ministry of Foreign Affairs.

[1] Current affiliation is: Technical University of Valencia / DSIC / ELP, Camino de Vera s/n, 46022 Valencia, Spain. Email: joubert@dsic.upv.es

doi:10.1016/j.entcs.2007.09.006

This promising scenario is mainly due to the advances in two major techniques: static program analysis and model checking. In a few words, a static analysis carries out a static (abstract) execution of a program in order to extract correct information about its behavior during execution. This information is typically utilized to discard some programming errors as soon as possible, and to improve the program compilation. Regarding model checking, it is a technique based on exhaustive exploration of states produced by (concurrent) programs, to check the satisfiability of a desirable program behavior. Our work exploits the use of model checking as a static analysis implementation method that can be naturally coupled with compilers.

One of the most well used static analyses is *data flow analysis* (DFA), which consists in giving the definition and usage of program data, such as variables, expressions, and definitions. This technique usually aims at compiler optimizations, *e.g.* dead code elimination, but it can also be used for state space reduction during explicit program verification, *e.g.* in software model checking, by keeping only necessary variables at each program point [2].

Example 1.1 We illustrate the classical *live variables* (*LV*) analysis on Figure 1. The goal of *LV* is to attach each program point (program counter) with the set of variables that may be live at this point [15]. A variable is live at a given point p iff its current value may be used, before being updated, by some program execution starting at p. The C code for the factorial problem on the left hand side of Figure 1

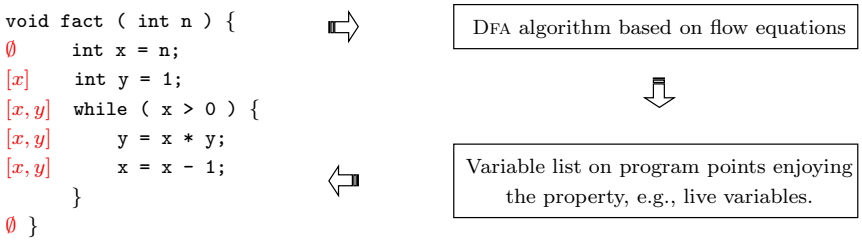

Fig. 1. Classical live variables analysis of a C program for the factorial problem

includes the intended result for the *LV* analysis. It shows which variables x or y are live on the different program points. For instance, variable y is not live at the first two lines of the program, because its value is updated before being read.

It has been shown that DFA is solvable using model checking techniques [17], namely using modal μ-calculus. Each analysis is specified as a particular temporal modal formula and then verified using standard model checkers, the output of the equivalent static analysis being the information generated by the model checker. As a consequence, all the necessary work to create a new DFA tool is reduced to write an adequate temporal formula.

Our approach follows the work of [17] to implement static analysis using model checking. However we focus on a method, which allows us to exploit the benefits of on-the-fly model checking, where the system is solved incrementally and at no time is the complete solution stored in memory. To this aim, this paper presents the use of *Boolean Equation System* (BES) as a means of specifying and solving pro-

gram analysis. We illustrate the BES specification capability by encoding classical DFAS [15]. The resulting DFAs are nicely implemented thanks to a generic algorithm, which evaluates the different BES encodings over a given program described as an implicit *Labeled Transition System* (LTS). The algorithm was integrated into the well-known CADP toolbox [8], and experiments confirmed that it worked efficiently for realistic examples.

Compared with related works, the main contribution of the paper is a method to achieve static analysis using *on-the-fly* model-checking. This work generalises the local BES resolution used for influence analysis [4,5], by introducing a program representation that enables a broader set of analyses than previously described, together with efficient resolution algorithms. Although static analysis is traditionally done with global methods, like in the proposals centered in modal μ-calculus [17], on-the-fly methods are of importance when dealing with realistic complex programs. Indeed, for millions lines projects, constructing and handling the program representation becomes a bottleneck, and dynamic solutions are useful during the design process. In our approach, both LTS and BES are constructed dynamically, thus saving the generation of unnecessary parts of both structures for the given analysis. The choice of BES as a unifying representation for static analysis problems is also motivated by its successful application to numerous verification problems, such as equivalence checking, partial order reduction, horn clause resolution, abstract interpretation, model checking and conformance test case generation [10]. Its resolution time and memory complexities being linear in the size of the program model [14] make the BES an appropriate and efficient way to solve DFAs. Another contribution of the paper is the encoding of forward static analyses (*e.g.*, available expressions and reaching definitions analyses) only in terms of forward operators (successor transition), whereas such analyses are defined in the literature using predecessor information.

The reminder of the paper is organized as follows. Section 2 defines the program abstract control flow graph (CFG) as an LTS. After a brief definition of alternation-free modal μ-calculus and BES, Section 3 gives the encodings of several classical DFAs, in terms of BESs. Section 4 presents in detail the on-the-fly DFA algorithm, and its modularity to solve different static analysis problems. Section 5 describes the architecture of the ANNOTATOR data flow analyser, together with experimental data evaluating its functionality on standard analysis problems. Finally, Section 6 summarizes the results and indicates directions for future work.

2 Problem representation

Data flow analyses work on the control flow graph of programs. In this work, we consider the LTS model, which is suitable for concurrent system descriptions, in particular for CFGs. An LTS is a tuple $M = \langle S, A, T, s_0 \rangle$, where S is a finite set of states, A is a finite set of actions, $T \subseteq S \times A \times S$ is the set of labeled transitions, and s_0 is the initial state. A transition $(s, a, s') \in T$, also noted as $s \xrightarrow{a} s'$, states that the system can move from s to s' by executing action a (s' is an

a-successor of s). In the sequel, we assume that states in S are program counters of the system to be analyzed, and actions in A are the basic program instructions, that is to say, boolean expressions, assignments of program variables with arithmetical expressions, assertions, and the invisible instruction τ.

Example 2.1 Figure 2(a) shows the C code for the factorial problem and Figure 2(b) illustrates the corresponding CFG in terms of an LTS P, where states store the program counter (an element of set $\{0, 1, 2, 3, 4, 5\}$). Actions in P are the boolean expressions and assignments in the code. We remark that, since only the CFG is kept in the LTS, the current value of program variables in each state is ignored.

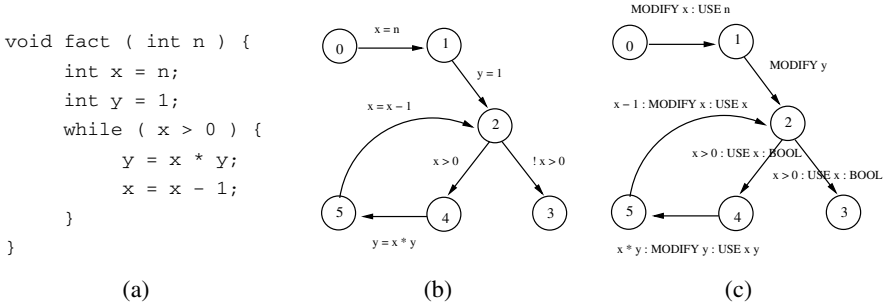

Fig. 2. C program for the factorial problem (a) with its original (b) and abstract (c) control flow graph

Let $M = \langle S, A, T, s_0 \rangle$ be a program CFG. In general, the goal of a DFA $dfs : S \to 2^D$ over M is to attach each program point $s \in S$ with a set of denotations $dfs(s) \in 2^D$ that correctly describe the program data at s during any execution. The type of denotations in D strongly depends on the analysis to be carried out. For instance, D could be the program variable set or the arithmetical and boolean expression set.

In order to make the resulting program description independent from the program specification language, we now define an abstraction of actions in the LTS to be analyzed. The proposed transformation extracts from each action the aspects relevant for the program property under analysis. By extracting more information out of the original program instructions, our CFG representation is less abstracted than the CFG model presented in [4], and allows the computation of a broader set of static analyses. In the new *abstract* LTS, each 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. In particular, with respect to the classical DFA problems, we are mainly interested in the set of program variables and expressions that are present in program instructions. Thus, we will use only two types of values, *var* and *expr*, denoting the set of program variables and expressions, respectively. Actions on abstract LTS transitions are of the form:

$$\vec{e} : MODIFY\ \vec{v} : USE\ \vec{w} : (BOOL|ASSERT|API)$$

and represent program instructions, where $\vec{e}$ is the list of non-trivial expressions, $\vec{v}$ is the list of modified variables in the instruction, and $\vec{w}$ is the list of used (*i.e.*, read)

variables. Moreover, three labels *BOOL*, *ASSERT*, and *API* respectively establish if the corresponding program instruction is a boolean or assignment expression, or a system call to an API of interest. This information is relevant for some static analyses, like influence analysis [2, 4, 5] and makes possible both property verifications and program optimizations. Such a format further allows to automatically and incrementally construct the (strict necessary parts of the) resulting LTS considering a program CFG.

Example 2.2 Figure 2(c) gives the complete abstract LTS corresponding to the factorial program presented on Figure 2(a) and 2(b).

Working at the level of an *abstract control flow graph* allows us to be independent from the input programming language and to focus on developing efficient analysis algorithms for it. Indeed, a same abstract CFG can represent a same program coded in different languages and it can also represent several programs that only differ from each other by arithmetic operators. This formalism enables the CFG representation of numerous high level programming languages, such as C, Promela, and LOTOS, and can serve as a benchmark representation for existing specialized analysis tools.

3 Data flow analysis as Boolean equation system

In this section, we introduce the Boolean equation system (BES) formalism and show that specifying DFAs with BESs allows us to construct a general and efficient approach to implement the analyses independently from the data properties being preserved (live variables, busy expressions, *etc.*). In a first step, we briefly recall how a DFA problem is equivalent to model checking modal μ-calculus formulas. Then, we introduce optimised BES resolutions as alternative and uniform techniques to solve on-the-fly DFAs.

3.1 Alternation-free modal μ-calculus model checking

The approach of computing data flow analyses by performing a transformation to μ-calculus formulas or systems of modal fixed point equations (DFA-MC) was proposed in [17], and was the basis of a verification component in jABC [11]. Formulas of alternation-free modal μ-calculus, noted L^1_μ, and defined over an alphabet of propositional variables $X \in \mathcal{X}$, have the following syntax given in positive form:

$$\phi \quad ::= \quad \mathsf{false} \mid \mathsf{true} \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \langle a \rangle\, \phi \mid [a]\, \phi \mid X \mid \mu X.\phi \mid \nu X.\phi$$

The semantics of a formula ϕ over an LTS $M = \langle S, A, T, s_0 \rangle$ denotes the set of states satisfying ϕ and it is defined as follows: boolean operators have their usual definition; possibility operator $\langle a \rangle\, \phi$ (necessity operator $[a]\, \phi$) define states from which some (all) transitions labeled by action a lead to states satisfying formula ϕ; minimal (maximal) fixed point operator $\mu X.\phi$ ($\nu X.\phi$) denotes the least (greatest) solution of the fixed point equation $X = \phi$, interpreted over domain 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.

Example 3.1 We illustrate the *LV* analysis using model checking on Figure 3. *LV* analysis can be defined by the following modal μ-calculus formula ϕ [17]: $\phi(v) = \mu Z.\ (\ \langle a \mid used(v,a)\rangle\ \mathsf{true}\ ) \vee (\langle a \mid \neg modified(v,a)\rangle\ Z)$, where $used(v,a)$ is true if variable v is used (*i.e.*, read) in instruction a, $bool(a)$ is true if instruction a is a boolean expression, and $modified(v,a)$ is true if variable v is modified (*i.e.*, defined) on instruction a. Given the factorial program on the left hand side of Figure 3,

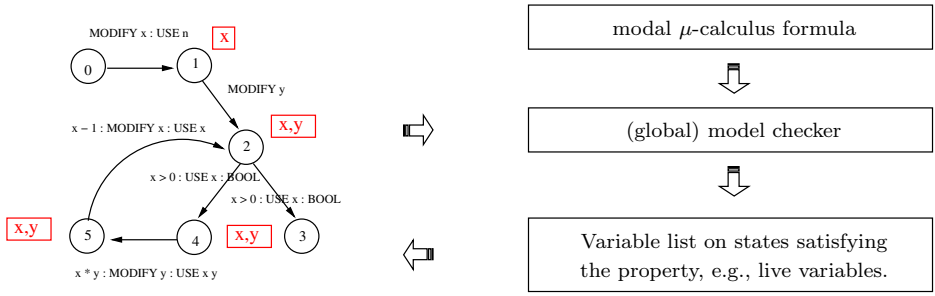

Fig. 3. Live variables analysis of the factorial abstract control flow graph using model checking

the analysis consists in evaluating on each state and for each program variable, the corresponding modal μ-calculus formula by a global model checker. The result shows which variables x or y are live on the different states.

Remark 3.2 Although the alternation-free fragment of modal μ-calculus is sufficient to describe all DFAs present in this article, we should notice that the value-based extension of L^1_μ [13] may be of interest when considering a broader set of DFAs, such as for influence analyses [4, 5].

3.2 Alternation-free boolean equation system resolution

A *Boolean Equation System* (BES) [1, 12] is a tuple $B = \langle x, M_1, \ldots, M_n\rangle$, 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 (maximal) fixed point equations with sign $\sigma_i = \mu$ ($\sigma_i = \nu$). 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 BES is *alternation-free* if there are no cyclic dependencies between its blocks. The *local* (or *on-the-fly*) resolution of an alternation-free BES $B = \langle x, M_1, \ldots, M_n\rangle$ consists in computing the value of x by exploring the right-hand sides of the equations in a demand-driven way, without explicitly constructing the blocks. Several on-the-fly alternation-free BES resolution algorithms with linear time and space complexity are available in the literature [13, 12].

To the best of our knowledge, no encodings of DFAs in terms of BES resolution have been proposed in the literature.

Example 3.3 Following the translation from state to Boolean formulas of Table 1 [14], the encoding of LV analysis in terms of BES is straightforward given the corresponding μ-calculus formula as expressed in Example 3.1. The least fixed point

Table 1
Translation from state to Boolean formulas

ϕ	$(\phi)_p$	$op(\phi)$	ϕ	$(\phi)_p$	$op(\phi)$
false	$\emptyset$	$\vee$	$\langle a \rangle \phi_1$	$\bigcup_{p \xrightarrow{a} q} (\phi_1)_p$	$\vee$
true		$\wedge$	$[a] \phi_1$		$\wedge$
$\phi_1 \vee \phi_2$	$(\phi_1)_p \cup (\phi_2)_p$	$\vee$	X	$\{X_p\}$	$\vee$
$\phi_1 \wedge \phi_2$		$\wedge$	$\sigma X.\phi_1$		$op(\phi_1)$

operator is stated explicitly by the boolean equation. Forward possibility modality (*i.e.*, $\Diamond$) operator translates into disjunction over all successor states. Boolean expressions over states translate into boolean expressions over actions. The resulting alternation-free BES given an LTS $M = \langle S, A, T, s_0 \rangle$ describing the program CFG is as follows: $X_{s,v} \stackrel{\mu}{=} \bigvee(\{\mathsf{true} \mid s \xrightarrow{a} s' \wedge used(v,a)\} \cup \{X_{s',v} \mid s \xrightarrow{a} s' \wedge \neg modified(v,a)\})$, where $s, s' \in S$, $a \in A$, and $v \in var$.

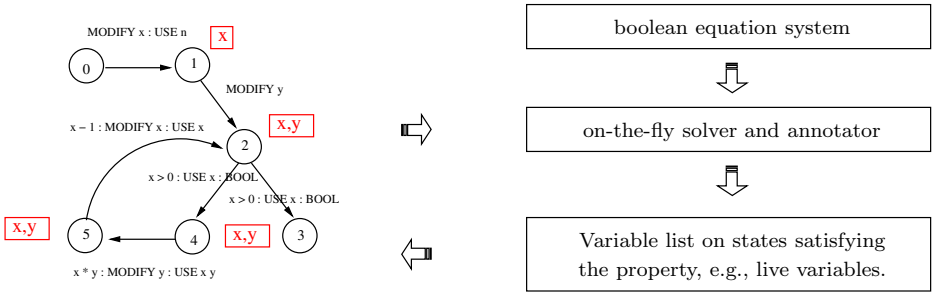

Fig. 4. Live variables analysis of the factorial abstract control flow graph using boolean equation system

Similarly to model checking μ-calculus formulas (Figure 3), the analysis on Figure 4 consists in solving for each state and program variable, the corresponding BES by an on-the-fly BES solver. If $X_{s,v}$ is true (false), then variable v is live (dead) at state s of the CFG, and the variable v is added to the set of live (dead) variables returned for state s, as shown by the annotations of the different states with variables x and y on Figure 4. Consequently, we directly obtain *dead variables* analysis by considering the negation of LV analysis.

Remark 3.4 As for modal μ-calculus formulas, we should notice that the parameterised extension of alternation-free BES [13], called PBES, may be of interest when considering further analyses, such as for influence analyses [4,5].

3.3 Very busy/available expressions and reachable definitions analyses

Here, we present the encodings of three classical data flow analyses, defined in [16,15], directly in terms of BESs (DFA-BES).

Let $M = \langle S, A, T, s_0 \rangle$ be an LTS representing the program abstract CFG to be analysed. Table 2 shows the BES encodings of the analysis between M and a program expression e or a program definition (o, v, t) with $v \in var$, modulo three widely-used DFAs: *very busy expressions* (VBE), *reaching definitions* (RD), and *available expressions* (AE) [15]. LV, VBE, AE and RD analyses are perhaps the most famous examples of flow analyses and are meant to portray backward and forward analyses with least and greatest fixed points.

Each analysis is represented as a BES with a single ν block defining, for example, for each couple of state and program expression $(s,e) \in S \times expr$, a variable $X_{s,e}$, which expresses that expression e satisfies the DFA on state s. We can notice that encodings given in Table 2 are based upon computing successors of states, therefore allowing to construct, on-the-fly, the LTS during BES resolution.

In this paper, we consider the commonly accepted standard definition of VBE analysis [15]. We first describe it as a modal μ-calculus formula: $\phi(e) = \nu Z.\ isUsed(e) \vee (\neg isModified(e) \ \wedge \Diamond\ \mathsf{true} \ \wedge \Box\ Z)$. Then, applying the transformations of Table 1, we give its encoding in terms of BES in Table 2. Even if $Y_{s,e}$ is a disjunctive variable, it has only one successor: true or false. Hence, like the LV analysis, which only uses disjunctions, we can apply an optimised BES resolution algorithm based on depth-first search for strictly conjunctive or disjunctive equation blocks, such as algorithm $A4$ of [14]. If $X_{s,e}$ is true, then expression e is very busy at state s of the CFG.

Contrary to backward analyses, such as LV or VBE analyses, forward DFAs calculate information about histories. Therefore, they are using backward modalities, which are overlined possibility (*i.e.*, $\bar{\Diamond}$) and necessity (*i.e.*, $\bar{\Box}$) operators. Such analyses cannot be encoded directly in terms of on-the-fly BES resolution, where only successor information is accessible. Moreover, there is no method to translate overlined modalities into forward modalities. Hence, instead of testing the final value of a computed boolean variable, *i.e.*, if $X_{s,e}$ is true then the property is satisfied on state s for element e, we resolve forward analyses with on-the-fly BES resolutions by computing a specific BES and by testing the inclusion of a boolean variable of interest in the set of computed variables.

We give a detailed description of such BES transformation for RD analysis. RD analysis computes for each program point, which assignments *may* have been made and not overwritten, when program execution reaches this point along some path [15]. We can reformulate this analysis as follows: Let the transition (o,v,t) be a *variable definition* to analyse, $v \in var$. To check if definition (o,v,t) is reachable on a given state s of the abstract CFG $M = \langle S, A, T, s_0 \rangle$, we traverse the graph until we encounter (o,v,t) (1). Then, we construct the reachable graph from state t, which does not overwrite definition (o,v,t) (2). Finally, if state s belongs to the reachable graph, then (o,v,t) is a RD on state s (3). Points (1) and (2) are expressed as two

Table 2
Boolean equation system encodings of three widely-used data flow analyses

Very busy expressions:

$$\left\{\begin{array}{l} X_{s,e} \stackrel{\nu}{=} \bigwedge(\{Y_{s,e}\} \cup \{\mathsf{true} \mid s \xrightarrow{a} s' \wedge used(e,a)\} \cup \\ \qquad\qquad \{\mathsf{false} \mid s \xrightarrow{a} s' \wedge \neg used(e,a) \wedge modified(e,a)\} \cup \\ \qquad\qquad \{X_{s',e} \mid s \xrightarrow{a} s' \wedge \neg modified(e,a) \wedge \neg used(e,a)\}) \\ Y_{s,e} \stackrel{\nu}{=} \bigvee(\{\mathsf{true} \mid s \xrightarrow{a} s'\}) \end{array}\right\} \begin{array}{l} s, s' \in S, \\ a \in A, \\ e \in expr \end{array}$$

Reaching definitions:

$$\left\{\begin{array}{l} X_{s,(o,v,t)} \stackrel{\nu}{=} \bigwedge(\{Y_{t,(o,v,t)} \mid s = o\} \cup \{X_{s',(o,v,t)} \vee \mathsf{true} \mid s \rightarrow s'\}) \\ Y_{s,(o,v,t)} \stackrel{\nu}{=} \bigwedge(\{Y_{s',(o,v,t)} \mid s \xrightarrow{a} s' \wedge (\neg modified(v,a))\}) \end{array}\right\} \begin{array}{l} s, s', o, t \in S, \\ a \in A,\ v \in var \end{array}$$

Available expressions:

$$\left\{\begin{array}{l} X_{s,e} \stackrel{\nu}{=} \bigwedge(\{Y_{s',e} \mid s \xrightarrow{a} s' \wedge ((\neg used(e,a) \wedge Z_{s,e} \notin computed(Z)) \\ \qquad\qquad \vee modified(e,a)) \wedge Y_{s,e} \notin computed(Y)\} \cup \\ \qquad\qquad \{Z_{s',e} \mid s \xrightarrow{a} s' \wedge used(e,a) \wedge \neg modified(e,a) \\ \qquad\qquad\qquad \wedge Z_{s,e} \notin computed(Z)\} \cup \\ \qquad\qquad \{X_{s',e} \vee \mathsf{true} \mid s \xrightarrow{a} s'\}) \\ Y_{s,e} \stackrel{\nu}{=} \bigwedge(\{Y_{s',e} \mid s \xrightarrow{a} s' \wedge (\neg used(e,a) \vee modified(e,a))\}) \\ Z_{s,e} \stackrel{\nu}{=} \bigwedge(\{Z_{s',e} \mid s \xrightarrow{a} s' \wedge \neg modified(e,a)\}) \end{array}\right\} \begin{array}{l} s, s' \in S, \\ a \in A, \\ e \in expr \end{array}$$

tautology boolean variables $X_{s,(o,v,t)}$ and $Y_{s,(o,v,t)}$ respectively that are always true and use a maximal fixed point to explore all possible paths, as shown on Table 2. Point (3) is achieved by testing the inclusion of $Y_{s,(o,v,t)}$ in the set of computed boolean variables in block Y ($computed(Y)$).

Example 3.5 Given the program abstract CFG of Figure 2(c), the result of computing RD analysis using our BES transformation is illustrated on Figure 5.

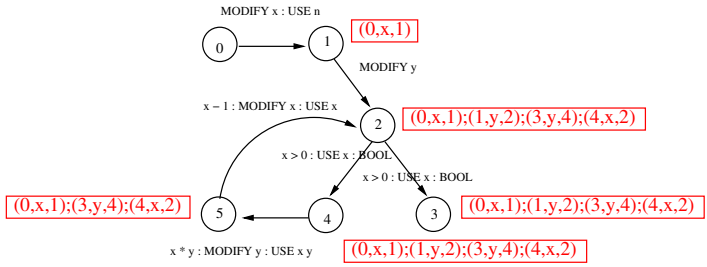

Fig. 5. Reaching definitions analysis of the factorial abstract CFG using boolean equation system

Following the same approach, AE analysis can be encoded in terms of the BES given in Table 2: block X computes for each successor state s' of current state s, either a subgraph of the LTS, where the expression is either unused or modified (*i.e.*, block Y), or a subgraph of the LTS where the expression is available (*i.e.*, block Z).

Since variable $X_{s,e}$ is recursively defined in terms of its successors $X_{s',e}$, solving $X_{s,e}$ forces to traverse the whole LTS. $X_{s,e}$ being a tautology, its final value is always true. Once $X_{s,e}$ is computed, testing the inclusion of $Y_{s,e}$ in $computed(Y)$ gives that expression e is not available at state s. If $Y_{s,e}$ is not included in $computed(Y)$, then testing the inclusion of $Z_{s,e}$ in $computed(Z)$ gives that expression e is available at state s. Otherwise, e is not available at s.

Considering $X_{s',e} \vee \mathsf{true}$ ($X_{s',(o,v,t)} \vee \mathsf{true}$) as an extra disjunctive boolean variable, an optimised depth-first search resolution algorithm for equation blocks containing alternating dependencies between conjunctive and disjunctive variables (*e.g.*, mode 5 of [14]) can be applied on the two later BESs.

4 On-the-fly data flow analysis algorithm

In order to simplify the presentation, we first discuss the case of LV analysis, and then we extend the method to the other static analyses. Thus, Figure 6(a) shows the implementation of LV on a program abstract CFG $M = \langle S, A, T, s_0 \rangle$ using BES resolution. We assume that we have already specified the data flow analysis for each program variable v by means of a BES $B = \langle X_{s,v}, M_1, \cdots, M_n \rangle$, as in Example 3.3, where s is a state of the LTS. We also assume that $solve(X_{s,v})$ computes the truth value of variable $X_{s,v}$ on the LTS M.

Function ANALYSE is based upon the following principle: starting at the initial state s_0, it performs an on-the-fly exploration of the LTS and, for each program state, computes the final value of boolean variable $X_{s,w}$ (*i.e.*, the result of function *solve*) considering each program variable w found during the algorithm execution (lines 23–27). If the final value of $X_{s,w}$ is true, we can conclude that DFA of state s with variable w is satisfied, that is to say, w is live at s. An annotating function d then keeps track of this result (line 25).

ANALYSE is more modular than algorithm INFLUENCE_ANALYSIS presented in [4], by allowing both the computation of classical DFAs and influence analyses. Moreover, it does not require an *a priori* computed set of program variables or expressions to be analyzed, since it dynamically constructs these sets on demand, with respect to a given analysis, while traversing the LTS.

We describe below the details of the algorithm. Visited but unexplored states are stored in set $visited \subseteq S$, whereas explored states are stored in set $explored \subseteq S$, $explored \cap visited = \emptyset$. The set of variables appearing at each action a in the LTS is denoted with $v(a)$, and added to set pv. At each iteration of the main loop (lines 5–30), a new state s is extracted from the set *visited*. In a first step, its unexplored successors are added to *visited*, and for each new variable $w \in v(a)$, the algorithm re-initiates the static analysis for all explored states s' by computing the boolean variable $X_{s',w}$ (lines 7–22). When all sets of states (*i.e.*, *visited* and *explored*) and variables (*i.e.*, pv) are updated, current state s is then analysed (lines 23–27). The algorithm terminates when all states have been explored. After termination of the main loop, the annotating function d, which was computed for all states and variables, is returned (line 31).

```
1. function ANALYSE (S,A,T,s0) : (S → 2^var) is
2.     var s, s' : S; d : S → 2^var; pv : 2^var; w : var;
3.         v : A → 2^var; visited, explored : 2^S;
4.     visited := {s0}; explored := ∅; pv := ∅;
5.     while visited ≠ ∅ do
6.         s := get(visited); d(s) := ∅;
7.         forall s -a-> s'' do
8.             if s'' ∉ explored then
9.                 visited := visited ∪ {s''}
10.            endif;
11.            if a ≠ τ then
12.                if v(a) \ pv ≠ ∅ then
13.                    forall s' ∈ explored,
14.                            w ∈ v(a) | w ∉ pv do
15.                        if solve(X_{s',w}) then
16.                            d(s') := d(s') ∪ {w}
17.                        endif
18.                    endfor
19.                endif;
20.                pv := pv ∪ v(a)
21.            endif
22.        endfor;
23.        forall w ∈ pv do
24.            if solve(X_{s,w}) then
25.                d(s) := d(s) ∪ {w}
26.            endif
27.        endfor;
28.        explored := explored ∪ {s};
29.        visited := visited \ {s}
30.    endwhile;
31.    return d
32. end
```

(a)

```
1. (S → 2^expr)
2. d : S → 2^expr; pe : 2^expr
3. z : expr; e : A → 2^expr
4. pe := ∅;
5.
6.
7.
8.
9.
10.
11.
12. e(a) \ pe ≠ ∅
13.
14. z ∈ e(a) | z ∉ pe
15. X_{s',z}
16. {z}
17.
18.
19.
20. pe := pe ∪ e(a)
21.
22.
23. z ∈ pe
24. X_{s,z}
25. {z}
26.
27.
28.
29.
30.
31.
32.
```

(b)

Fig. 6. On-the-fly program variables (a) or expressions (b) analysis of LTS $M = \langle S, A, T, s_0 \rangle$ using BES resolution of $X_{s,w}$ (a) or $X_{s,z}$ (b)

Figure 6(b) shows the extension and/or modification of the algorithm on Figure 6(a) for analyzing program expressions. The main difference between both versions is the progressive construction of a set pe of arithmetical expressions extracted by $e(a)$ from the actions encountered during the LTS traversal.

By replacing variable $X_{s,w}$ ($X_{s,z}$) in function ANALYSE by $X_{s,v}$ ($X_{s,e}$), defining LV (VBE) as a BES in Section 3, we immediately obtain an on-the-fly LV (VBE) analysis algorithm. As a result, backward DFAs can be systematically encoded in terms of BESs and solved by ANALYSE.

In addition, the same algorithm can also be used for other static analyses modulo a modification of the output signature, because the algorithm is only an intertwined traversal of the LTS for each program variable or expression. For instance, we can slightly modify the algorithm ANALYSE on Figure 6(a) by taking into account *variable definitions*, which are triple of state, variable and state (o,v,t), instead of considering only program variables returned by $v(a)$ (lines 14 and 20). By redefining $pv : 2^{S\times var\times S}$, $v : A \to 2^{S\times var\times S}$, and $d : S \to 2^{S\times var\times S}$, we directly obtain an on-the-fly *RD* analysis by solving $X_{s,w}$ (lines 15 and 24) and by testing the inclusion of $Y_{s,w}$ in $computed(Y)$.

We discuss below the behavior of algorithm ANALYSE w.r.t. efficiency. Every transition in the LTS is traversed exactly once per program variable or expression. We assume that BES resolutions are linear in the size of the LTS [14]. Since the constructed BES is unique for all states given a variable or an expression, resolution of already solved boolean variables is done in constant time. Therefore, each call to ANALYSE has a worst-case time complexity $O(|S|+|T|)$, considering that the number of tested variables and expressions is significantly smaller than the number of states and transitions. The same bound applies for memory consumption, since in the worst case, every state will be stored in set *explored*, taking in account that BES resolution has a linear memory complexity.

Table 3
Comparison of three data-flow analysis techniques

	Classic DFA	DFA-MC	DFA-BES
Program representation	control flow graph	Kripke/transition system	labeled transition system
Problem statement	flow equation system over sets	L^1_μ formula	BES
Computation of the solution	DFA algorithm (adhoc)	(global) model checker (generic)	*on-the-fly* BES solver + annotator (generic)

As shown by Table 3, the solution using BES resolution is similar in spirit to the model checking approach, as it enables to directly provide the desired property as an equation system instead of a μ-calculus formula. BESs being solved on-the-fly, only relevant program parts are computed for each state, variable or expression. Another advantage of working at the level of BESs is that persistent computation results between subsequent resolution calls can be used to obtain an efficient overall resolution. Indeed, only one structure, the boolean equation system, is computed for a given analysis and variable of interest, whereas on-the-fly evaluation of μ-calculus formulas would compute one formula per state of the LTS, as it is done in [4].

5 Implementation and experiments

We implemented the ANALYSE algorithm described in Section 4 in a modular tool, called ANNOTATOR, which is built within CADP [8] upon the primitives of the OPEN/CÆSAR [7] environment for on-the-fly exploration of LTSs and on-the-fly resolution of BESs. Currently, ANNOTATOR achieves four influence analyses [4, 5] and the five DFAs described in Section 3. We briefly describe the architecture of this tool and give some experimental results concerning program analyses.

5.1 The ANNOTATOR tool

The static analyser ANNOTATOR (see Figure 7) takes as input the LTS associated to the program abstract CFG together with optional arguments such as the type of the underlying BES resolution (*e.g.*, breadth-first search), and produces as output the function d as an XML or textual file.

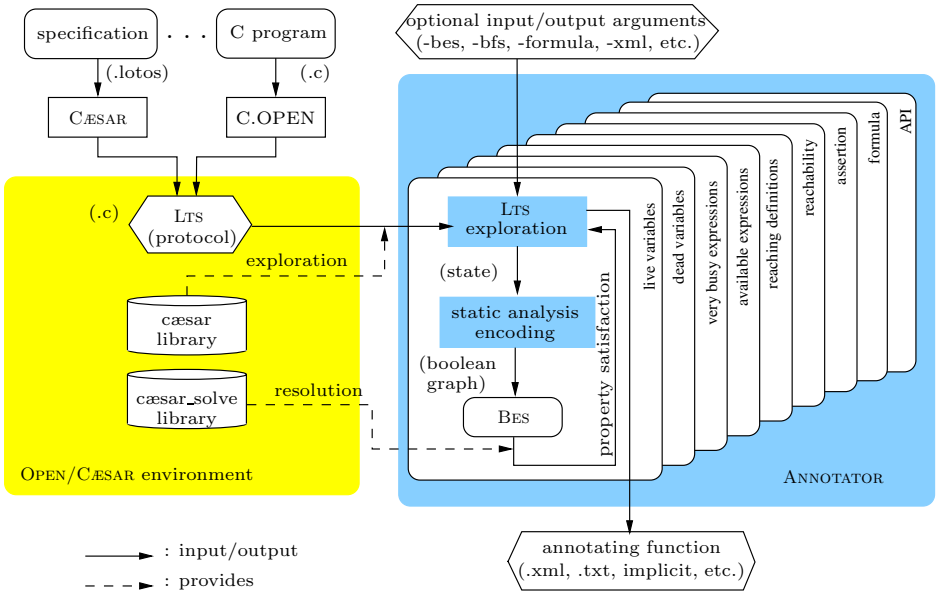

Fig. 7. The on-the-fly ANNOTATOR tool

ANNOTATOR consists of two parts: a front-end, responsible for encoding the static analysis of LTS as a (parameterised) BES resolution, and a back-end, responsible of (parameterised) BES resolution, playing the role of verification engine. Back-end is obtained by using algorithms of the CÆSAR_SOLVE library [14]. Globally, the approach to on-the-fly static analysis is both to construct on-the-fly the LTS and corresponding (parameterised) BES and to determine the final value of variables of interest specified by ANALYSE. Only the part of both graphs that is necessary to perform the static analysis is explored incrementally.

Our resulting data flow analyser can also be slightly extended to become an OPEN/CÆSAR library providing an Application Programming Interface (API) allowing the on-the-fly use of computed function d. Hence, (existing) compilers and

verification tools would directly be able to integrate on-the-fly static analysis into their computation without needing an exhaustive explicit intermediate format (*e.g.*, XML).

The implementation of the tool including all analyses was done rapidly thanks to the appropriateness of OPEN/CÆSAR APIs to our DFA problem encoding. Each BES description is about 600 lines of C code. They are all used from a common part, namely the algorithm ANALYSE, which is about 2 000 lines of C code. Currently implemented analyses, accessible through command line parameters, should be sufficient for a basic use of ANNOTATOR. In order to extend the current tool with new analyses, one can think of three possibilities: first, a user could (graphically) specify the expected analysis as an implicit BES, from which the tool would automatically generate the corresponding analysis module (following the same approach as [18]). Another possibility is to write the analysis as a μ-calculus formula, translated by the tool into a BES (following the approach of [14]). Otherwise, the description of the BES in a C file can also be considered. We expect that it should be possible to construct new analyses of comparable complexity in a matter of hours, now that the infrastructure is stable.

5.2 Experimentation

We performed several experiments to compare the results of our DFAs with those observed in the literature [15]. We considered twelve classical C program examples showing the interest of using one or the other DFA to simplify the compilation of the program. We also performed a series of experiments for investigating the effectiveness of influence analysis [4, 5]. Besides Promela examples extracted from the literature [2], we also considered ten other C program examples specific for each one of the implemented influence analyses.

All abstract CFGs, extracted from the examples, were explicitly described thanks to the *Binary Coded Graph* (BCG) [8] format. Since their sizes were rather small, experiments returned immediate results allowing us to verify only functionality properties of our prototype, but neither time nor memory statistics.

To show the applicability of our tool to a third specification language, we took a LOTOS description of the Dekker mutual exclusion protocol [2] (89 lines of LOTOS, 2 processes, 9 variables, 954 states, 1 908 transitions, and 17 labels) on which we tested the nine currently implemented DFAs. The first step was to automatically extract out of the specification a specific Petri net, called *network*, that can be further processed to obtain a CFG of the protocol (25 states, 134 transitions) [9]. The graph was then transformed into our abstract model, by appropriately relabeling the transitions. Finally, the resulting abstract LTS was processed by ANNOTATOR. From the results, we conclude that, among the nine variables present in the specification, none of them were live at all states. This means that an explicit verification of the program can ignore many variables in the state vector at different control point without loosing any information relevant for a formula to be evaluated (as

[2] http://www.inrialpes.fr/vasy/cadp/demos

it is done with *reset variables* analysis [9]). With respect to reaching definitions analysis, 162 program definitions were tested. We have observed that numerous definitions are reaching a majority of the states. Hence, using the set of reaching definitions, a compiler should perform an efficient constant propagation. Finally, among the ten tested program expressions, none of them were neither available at any state, nor very busy at states different from their first use. As a consequence, common sub-expressions cannot be eliminated in the protocol, and code hoisting would be useless. To show the scalability of ANNOTATOR, we have been successfully experimenting the tool on very large CFGs, extracted from the VLTS benchmark [3], with size up to 10^6 program counters and instructions, such as the *vasy_65_2621* LTS (65 537 states, 2 621 480 transitions). Finally, the ANNOTATOR tool is available on http://www.lcc.uma.es/∼joubert/software.html#ANNOTATOR together with result files for all examples, including the Dekker mutual exclusion protocol.

6 Conclusion and future work

We presented encodings of four classical data flow analyses in terms of BES resolution, and we automatized the program analysis process in conjunction with on-the-fly verification tools. The on-the-fly static analyser ANNOTATOR was developed using the OPEN/CÆSAR environment [7] of the CADP [8] toolbox. It implements static analyses of an application-independent representation of programs given as an LTS. The tool currently offers nine DFAs using optimised BES resolution algorithms. The experiments carried out using this tool on numerous standard examples assess the functionality of the static analysis, and demonstrate that the modular architecture of the tool allows a rapid integration of new static analyses described as BES resolutions, and a quick connection to existing compilers.

Although our initial goal was to extend compilers with data flow analysis algorithms based on model checking, a future work will consist in directly applying the technique on the compiler for certifying it. We also plan to continue our work along two additional directions. First, we would like to show the impact of automatic abstract matching on the explored state space size during verification of programs described in C code, thanks to the compiler C.OPEN proposed in [6]. To this purpose, the static analysis resulting from the interconnection of ANNOTATOR and C.OPEN into CADP could then be integrated to the model extractor SOCKETMC [3]. Finally, we will seek solutions to other static analysis problems, such as *reset variables* analysis [9], by investigating their translation in terms of BES resolution, and by studying the *a priori* natural and efficient connection of existing compilers to our abstract CFG model.

References

[1] Andersen, H. R., *Model checking and boolean graphs*, Theoretical Computer Sci. **126** (1994), pp. 3–30.

[3] http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html

[2] Cámara, P., M. Gallardo and P. Merino, *Abstract matching for software model checking*, in: A. Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking of Software SPIN'06 (Vienna, Austria)*, LNCS **3925** (2006), pp. 182–200.

[3] Camara, P., M. Gallardo, P. Merino and D. Sanán, *Model checking software with well-defined apis: the socket case*, in: S. Gnesi, T. Margaria and M. Massink, editors, *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2005 (Lisbon, Portugal)* (2005), pp. 17–26.

[4] Gallardo, M., C. Joubert and P. Merino, *Implementing influence analysis using parameterised boolean equation systems*, in: *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation ISOLA'06 (Paphos, Cyprus)* (2006).

[5] Gallardo, M., C. Joubert, P. Merino and D. Sanán, *On-the-fly API influence analysis of software*, in: P. Merino and M. Bakkali, editors, *Proceedings of the 2nd International Conference on Science and Technology JICT'07 (Málaga, Spain)* (2007).

[6] Gallardo, M., P. Merino and D. Sanán, *Towards model checking c code with open/cæsar*, in: J. Barjis, U. Ultes-Nitsche and J. C. Augusto, editors, *Proceedings of the 4th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems MSVVEIS'2006 (Paphos, Cyprus)* (2006), pp. 198–201.

[7] Garavel, H., *Open/cæsar: An open software architecture for verification, simulation, and testing*, in: B. Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, LNCS **1384** (1998), pp. 68–84.

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

[9] Garavel, H. and W. Serwe, *State space reduction for process algebra specifications*, Theoretical Computer Science **351** (2006), pp. 131–145.

[10] Joubert, C. and R. Mateescu, *Distributed on-the-fly model checking and test case generation*, in: A. Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking of Software (SPIN'06)*, LNCS **3925** (2006), pp. 126–145.

[11] Lamprecht, A.-L., T. Margaria and B. Steffen, *Data-flow analysis as model checking within the jabc*, in: A. Mycroft and A. Zeller, editors, *Proceedings of the 15th International Conference on Compiler and Construction CC'2006 (Vienna, Austria)*, LNCS **3923** (2006), pp. 101–104.

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

[13] Mateescu, R., *Local model-checking of an alternation-free value-based modal mu-calculus*, in: A. Bossi, A. Cortesi and F. Levi, editors, *Proceedings of the 2nd International Workshop on Verication, Model Checking and Abstract Interpretation VMCAI'98 (Pisa, Italy)*, University Ca' Foscari of Venice, 1998.

[14] Mateescu, R., *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** (2006), pp. 37–56.

[15] Nielson, F., H. Nielson and C. Hankin, "Principles of Program Analysis," 2005.

[16] Schmidt, D. A., *Data flow analysis is model checking of abstract interpretations*, in: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages POPL'98 (San Diego, USA)* (1998), pp. 38–48.

[17] Steffen, B., *Data flow analysis as model checking*, in: T. Ito and A. R. Meyer, editors, *Proceedings of the International Conference on Theoretical Aspects of Computer Software TACS'91 (Sendai, Japan)*, LNCS **526** (1991), pp. 346–365.

[18] Steffen, B., *Generating data flow analysis algorithms from modal specifications*, Science of Computer Programming **21** (1993), pp. 115–139.