On-the-Fly API Influence Analysis of Software ⋆

María del Mar Gallardo [a], Christophe Joubert [b], Pedro Merino [a] and David Sanán [a]

[a] *University of Málaga, Campus de Teatinos s/n, 29071, Málaga, Spain*

[b] *Technical University of Valencia, Campus de Vera s/n, 46022, Valencia, Spain*

Abstract

In order to combat the state space explosion resulting from explicit-state model checking of software, we investigate the use of a *parameterised boolean equation system* (PBES) to solve on-the-fly (*i.e.*, with incremental construction of the program state space) influence analysis of program variables *w.r.t. Application Programming Interface* (API) calls executed in the program. The static analysis results are then processed to simplify the program state vector by keeping only program variables preserving all reachable API calls. Using the connection of the C compiler C.OPEN to the static analyser ANNOTATOR, we illustrate the benefit of such an analysis by reducing the state space of the Peterson mutual exclusion protocol, in which shared memory accesses are made through an API.

Key words: formal method, static analysis, software model checking, labeled transition system, boolean equation system

1. Introduction

Explicit-state verification of software, and especially distributed software, is prone to the state space explosion problem, due to complex (and infinite) data structures as well as (asynchronous) processes interleaving. Efficient abstraction and reduction techniques have been encountered during the last years, one of them, called *abstract matching* [1], consisting in abstracting the state vector of a program by keeping only relevant information *w.r.t.* to a class of properties to be verified. One implementation of such a technique, called *influence analysis* [2], extracts program variables in each program control point that preserve the reachable code dealing with properties of interest. In this paper, we introduce a new *on-the-fly* (*i.e.*, during the incremental construction of the program state space) influence analysis that enables the reduction of the program state vector while preserving properties depending on program API calls. We first describe the influence analysis (IA) problem in Section 2 and then give an extension to API preservation in Section 3 in terms of flow equations, *value-based alternation-free* MCL *formula* (L^1_μ), and *parameterised boolean equation system* (PBES). Section 4 shows details of implementation and experimentation on a C implementation of the Peterson mutual exclusion protocol. Finally, Section 5 gives some concluding remarks and future work.

⋆ This work has been supported by the Spanish MEC under grant TIN2004-7943-C04

Email addresses: gallardo@lcc.uma.es (María del Mar Gallardo), joubert@dsic.upv.es (Christophe Joubert), pedro@lcc.uma.es (Pedro Merino), sanan@lcc.uma.es (David Sanán).

2. Influence analysis

The static analysis method called influence analysis extracts the least set of significant variables for each program point *w.r.t.* to properties that have to be preserved in the program. In [2], the flow equation definitions of three influence analyses were given:

- $\mathsf{IA}_{reachability}$ preserves information on reachable code. The authors also gave an extended version of this analysis considering global variables;
- $\mathsf{IA}_{assertion}$ produces bigger sets of variables, but preserves *safety properties*. It extends $\mathsf{IA}_{reachability}$ considering variables contained in assertions; and
- $\mathsf{IA}_{formula}$ is the least precise analysis, but in contrast, it preserves *liveness properties*. It is based on considering as influential variables all variables appearing in the temporal formulas to be verified.

In [3], the authors gave a translation of the influence analysis problem in terms of L^1_μ formulas [4] and in terms of PBESs [4]. These formalisms allowed to combine on-the-fly techniques with compact program representation. The program control flow graph (CFG) was first abstracted into a *la-*

belled transition system (LTS) and then analysed by a general-purpose model checker or by a BES solver. This translation from flow equations to L^1_μ formulas and PBESs was extended to general purpose data flow analyses (DFAs) in [5].

Here, we follow the approaches of [3] and [2]: we first give a definition of the API *influence analysis* ($\mathsf{IA}_{\mathrm{API}}$) in terms of flow equations and then we translate it in terms of value-based L^1_μ formulas and PBES.

3. Extending influence analysis to APIs

A program variable is influential at a program state *w.r.t.* to APIs, when it satisfies any of the following conditions:
- the variable is used in an API call;
- the variable modifies a variable later used in an API call; or
- the variable modifies a variable used in a boolean expression.

This analysis preserves information on the reachable code and further considers the dependencies between program variables and API calls. For instance, this analysis is useful for verifying properties on the correct use of APIs in a program. The analysis result can further be processed to reduce the reachable program state space by excluding non-influential variables from the state vector.

3.1. *Flow equations*

$\mathsf{IA}_{\mathrm{API}}$ is a *backwards must* data flow analysis. The analysis is done in a backwards order, and the data flow confluence operator is set union. Using the notation of [6], the data flow equations used for a given instruction block s in $\mathsf{IA}_{\mathrm{API}}$ are:

$$\mathrm{IA}_{\mathrm{API,in}}[s] = \mathrm{GEN}[s] \cup (\mathrm{IA}_{\mathrm{API,out}}[s] - \mathrm{KILL}[s] \cup \mathrm{INFL}[s])$$

$$\mathrm{IA}_{\mathrm{API,out}}[final] = \emptyset$$

$$\mathrm{IA}_{\mathrm{API,out}}[s] = \bigcup_{api,bool \notin s, p \in succ[s]} \mathrm{IA}_{\mathrm{API,in}}[p]$$

$$\mathrm{GEN}[y_1, \cdots, y_n := bool(x_1, \cdots, x_n)] = \{x_1, \cdots, x_n\}$$

$$\mathrm{GEN}[y_1, \cdots, y_n := api(x_1, \cdots, x_n)] = \{x_1, \cdots, x_n\}$$

$$\mathrm{KILL}[(y_1, \cdots, y_n) := f(x_1, \cdots, x_n)] = \{y_1, \cdots, y_n\}$$

$$\mathrm{INFL}[(y_1, \cdots, y_n) := f(x_1, \cdots, x_n)] =$$

$$\textbf{if } (\{y_1, \cdots, y_n\} \cap \mathrm{IA}_{\mathrm{API,out}}[s] \neq \emptyset)$$

$$\textbf{then } \{x_1, \cdots, x_n\}$$

$x_1, \cdots, x_n$ and $y_1, \cdots, y_n$ are variables in *var* the set of program variables. $\vec{y} := f(\vec{x})$ is a program instruction block, where f is a functional using variables in $\vec{x}$, and where $\vec{y}$ gets modified by f. *bool* and *api* are similar functionals, but they respectively describe a boolean expression and an instruction call to an API.

We illustrate such analysis on a portion of C code extracted from a Peterson mutual exclusion [7] implementation using a shared memory API (*e.g.*, *sread* call):

```
pid = (pid + 1) % 2;
while (sread (flag1_des) == 1
      && sread (turn_des) == 1)
  printf (``Process %d waiting\n'', pid);
```

From the above flow equation definition, the $\mathsf{IA}_{\mathrm{API}}$ returns that variables *flag1_des* and *turn_des* are influential on every program point, whereas variable *pid* is not influential on any point.

The correctness of the $\mathsf{IA}_{\mathrm{API}}$ follows the same proof scheme as detailed in [2] for influence analyses $\mathsf{IA}_{reachability}$, $\mathsf{IA}_{assertion}$ and $\mathsf{IA}_{formula}$.

3.2. *Value-based L^1_μ formula and* PBES

When translating the problem of $\mathsf{IA}_{\mathrm{API}}$ from flow equations to MCL formulas, we also translate the problem of modeling the program control flow graph from textual specific description to implicit independent formalism such as LTS.

Using previous work on translating influence analysis problems into value-based L^1_μ formulas and PBESs [3], we can describe the problem of $\mathsf{IA}_{\mathrm{API}}$ by a least fixed point of a functional over all program states.

Given a CFG described as an LTS $M = \langle S, A, T, s_0 \rangle$, and a variable $v \in var$, Table 1 gives the encoding in terms of value-based L^1_μ formula of $\mathsf{IA}_{\mathrm{API}}$ variable v on all states of M.

In addition to $used(v, a)$ and $modified(v, a)$ primitives (indicating if a variable v is used (modified) by action a), we introduce $bool(a)$ ($api(a)$), which tests if a is a boolean (API) instruction.

In the table, the $\mathsf{IA}_{\mathrm{API}}$ formula is translated into a PBES with single μ block and parameter v of type *var* defining, for each couple of state and variable

Value-based μ-calculus formula:

$$\phi = \mu Y(v : var).(\ \langle a \mid used(v,a) \wedge (bool(a) \vee api(a))\rangle\ \mathsf{true} \vee \langle a \mid modified(z,a) \wedge used(v,a)\rangle\ Y(z) \vee \langle a \mid \neg modified(v,a)\rangle\ Y(v))$$

Parameterised boolean equation system:

$$\left\{ X_{s,v} \stackrel{\mu}{=} \bigvee(\{\mathsf{true} \mid s \xrightarrow{a} s' \wedge used(v,a) \wedge (bool(a) \vee api(a))\} \cup \{X_{s',z} \mid s \xrightarrow{a} s' \wedge modified(z,a) \wedge used(v,a)\} \cup \{X_{s',v} \mid s \xrightarrow{a} s' \wedge \neg modified(v,a)\}) \right\} s, s' \in S,\ a \in A,\ v, z \in var$$

Table 1. Value-based alternation-free μ-calculus formula and PBES encodings of API influence analysis

$(s,v) \in S \times var$, a variable $X_{s,v}$ which expresses that variable v is influential at state s [3]. Generalising the analysis to all program states is done via algorithm ANALYSE from [5].

The PBES shown in Table 1 can be solved using an optimised BES resolution algorithm based on depth-first search for disjunctive equation blocks, such as algorithm $A4$ of [8]. Here, the transformation of PBES into BES is direct, since the parameter v is part of the boolean variable definition. Hence, at most $|var|$ boolean variables will need to be solved for each state of the abstract CFG described as an LTS, before the analysis be terminated.

Using the same C implementation of Peterson mutual exclusion protocol as in Section 3.1, we illustrate on Figure 1 the construction of an $\mathsf{IA}_{\mathrm{API}}$ BES (lower part of Figure 1) on the program CFG given as an LTS (upper part of Figure 1). This BES intends to answer to the following question: "Does program variable *pid* influences state 0 of the program CFG?". Solving the BES returns that variable *pid* is false, hence it does not influence state 0 of the CFG. Further computations on all states of the CFG would finally give us (with algorithm ANALYSE) that variable *pid* is not influencing any state of the CFG, hence it can be excluded from the program state vector.

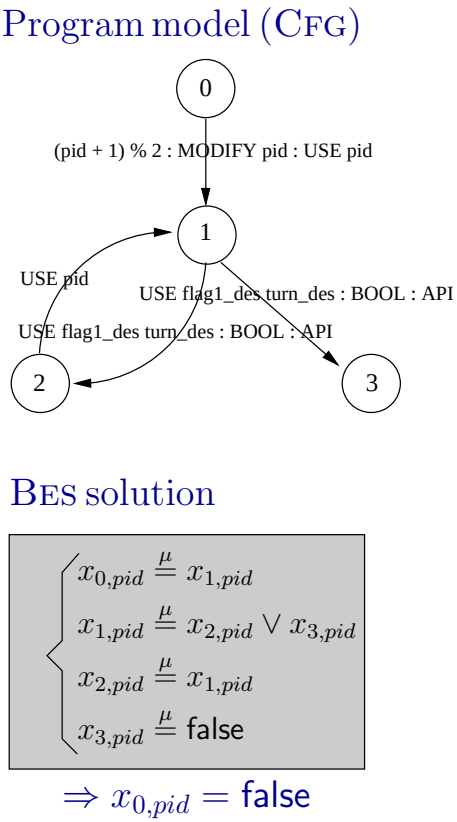

Fig. 1. Subset of the Peterson CFG and BES for the API influence analysis of variable *pid* on state 0

4. Implementation and experiments

We implemented the $\mathsf{IA}_{\mathrm{API}}$ PBES in our modular static analyser, called ANNOTATOR, which is built within CADP [9] upon the primitives of the OPEN/CÆSAR [10] environment for on-the-fly exploration of LTSs and on-the-fly resolution of BESs. Currently, ANNOTATOR achieves four influence analyses [3] and four classical DFAs [5].

The static analyser ANNOTATOR (see Figure 2) takes as input the LTS associated to the program

abstract CFG provided by C.OPEN and the type of analysis to carry out. It produces as output static analysis results (as XML or textual file) that can be further processed by the C.OPEN compiler to produce, for instance, smaller program state spaces. The original explicit state space of Peterson protocol contained 35 671 states and 57 066 transitions. After analysing the Peterson CFG with ANNOTATOR, our C compiler could reduce the Peterson state space to 25 655 states and 40 493 transitions. Further minimisations gave us a final state space of size 652 states and 1 255 transitions [1] .

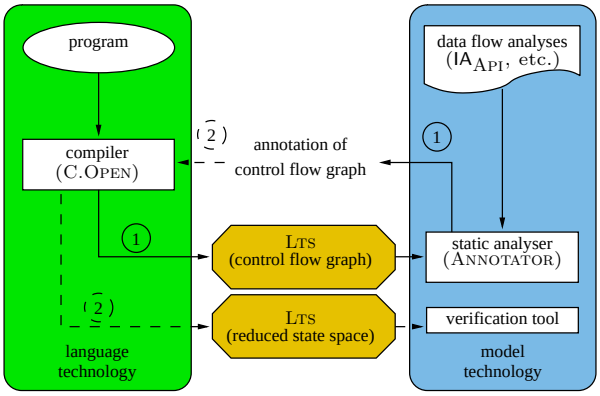

Fig. 2. The on-the-fly software state space construction (C.OPEN) and API influence analysis (ANNOTATOR) tools

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 [8]. 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 boolean variables of interest. Only the part of both graphs that is necessary to perform the static analysis is explored incrementally.

5. Conclusion and future work

Explicit-state software model checking requires techniques to abstract and reduce a program state space. Here, we presented a new influence analysis that preserves properties on program API calls. We gave encodings in terms of flow equations, value-based alternation-free MCL formula, and parameterised boolean equation system. Experiments on the Peterson mutual exclusion protocol showed important reduction of the program state space. An interesting line of research would be to combine different static analyses on the compiler side to further reduce the program state space *w.r.t.* specific properties of interest.

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

References

[1] G. J. Holzmann and R. Joshi. Model-driven software verification. In *Proc. of SPIN'04*, LNCS 2989, pp. 76–91.

[2] P. Cámara, M. Gallardo and P. Merino. Abstract matching for software model checking. In *Proc. of SPIN'06*, LNCS 3925, pp. 182–200.

[3] M. Gallardo, C. Joubert and P. Merino. Implementing influence analysis using parameterised boolean equation systems. In *Proc. of ISOLA'06*, IEEE Computer Society Press.

[4] R. Mateescu. Vérification des propriétés temporelles des programmes parallèles. Thèse de doctorat, Institut National Polytechnique de Grenoble, 1998.

[5] M. Gallardo, C. Joubert and P. Merino. On-the-fly data flow analysis based on verification technology. In *Proc. of COCV'07*, ENTCS.

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

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

[8] 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.

[9] 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.

[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.