Science of Computer Programming 93 (2014) 154–182

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions ☆, ☆☆

A. Miné [a,b,*]

[a] CNRS, France
[b] École normale supérieure, France

HIGHLIGHTS

- We automatically compute sufficient conditions for program correctness.
- We rely on Abstract Interpretation to compute sound approximate solutions.
- We present under-approximations for existing numeric abstract domains.
- We built a (freely available) prototype implementation using polyhedra.

ARTICLE INFO

Article history:
Received 4 February 2013
Received in revised form 31 July 2013
Accepted 23 September 2013
Available online 10 October 2013

Keywords:
Abstract interpretation
Backward analysis
Numeric abstract domains
Static analysis
Under-approximation

ABSTRACT

In this article, we discuss the automatic inference of sufficient preconditions by abstract interpretation and sketch the construction of an under-approximating backward analysis. We focus on numeric properties of variables and revisit three classic numeric abstract domains: intervals, octagons, and polyhedra, with new under-approximating backward transfer functions, including the support for non-deterministic expressions, as well as lower widenings to handle loops. We show that effective under-approximation is possible natively in these domains without necessarily resorting to disjunctive completion nor domain complementation. Applications include the derivation of sufficient conditions for a program to never step outside an envelope of safe states, or dually to force it to eventually fail. We built a proof-of-concept prototype implementation and tried it on simple examples. Our construction and our implementation are very preliminary and mostly untried; our hope is to convince the reader that this constitutes a worthy avenue of research.

1. Introduction

A major problem studied in program verification is the automatic inference of program invariants using forward analyses, as well as the inference by backward analysis of necessary conditions for programs to be correct. In this article, we consider the dual problem: the inference by backward analysis of *sufficient conditions* for programs to be correct.

Motivation As motivating example, consider the simple loop in Fig. 1(a): `j` starts from a random value in $[0; 10]$ and is incremented by a random value in $[0; 1]$ at each iteration. A forward invariant analysis would find that, at the end of the

☆ This work is supported by the INRIA project "Abstraction" common to CNRS and ENS in France.
☆☆ This work is an extended version of the article [1] published in the Proceedings of the 4th International Workshop on Numerical and Symbolic Abstract Domains (NSAD 2012).
* Correspondence to: Département d'informatique, École normale supérieure, 45 rue d'Ulm, F-75230 Paris Cedex 05, France. Tel.: +33 1 44 32 21 17.
E-mail address: mine@di.ens.fr.
URL: http://www.di.ens.fr/~mine.

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.09.014

```
(1) j = [0;10];
(2) i = 0;
(3) while (4) (i < 100) {
      (5) i++;
      (6) j = j + [0;1];
      (7)
}
(8) assert (j <= 105);
(9)
```

(a)

$$
\begin{aligned}
X_1 &= I\\
X_2 &= \tau\{\!\{\, j := [0;10] \,\}\!\}\, X_1\\
X_3 &= \tau\{\!\{\, i := 0 \,\}\!\}\, X_2\\
X_4 &= X_3 \cup X_7\\
X_5 &= \tau\{\!\{\, i < 100 \,\}\!\}\, X_4\\
X_6 &= \tau\{\!\{\, i := i + 1 \,\}\!\}\, X_5\\
X_7 &= \tau\{\!\{\, j := j + [0;1] \,\}\!\}\, X_6\\
X_8 &= \tau\{\!\{\, i \geq 100 \,\}\!\}\, X_4\\
X_9 &= \tau\{\!\{\, j \leq 105 \,\}\!\}\, X_8
\end{aligned}
$$

(b)

$$
\begin{aligned}
X_1 &= \overleftarrow{\tau}\{\!\{\, j := [0;10] \,\}\!\}\, X_2\\
X_2 &= \overleftarrow{\tau}\{\!\{\, i := 0 \,\}\!\}\, X_3\\
X_3 &= X_4\\
X_4 &= \overleftarrow{\tau}\{\!\{\, i < 100 \,\}\!\}\, X_5 \cap\\
&\quad\ \overleftarrow{\tau}\{\!\{\, i \geq 100 \,\}\!\}\, X_8\\
X_5 &= \overleftarrow{\tau}\{\!\{\, i := i + 1 \,\}\!\}\, X_6\\
X_6 &= \overleftarrow{\tau}\{\!\{\, j := j + [0;1] \,\}\!\}\, X_7\\
X_7 &= X_4\\
X_8 &= \overleftarrow{\tau}\{\!\{\, j \leq 105 \,\}\!\}\, X_9
\end{aligned}
$$

(c)

Fig. 1. A simple loop program (a), its concrete invariant equation system (b), and its concrete sufficient condition equation system (c).

loop, $\mathtt{j} \in [0; 110]$ and the assertion at line (8) can be violated. We are now interested in inferring conditions on the initial states of the program assuming that the assertion actually holds. A backward analysis of necessary conditions would not infer any new condition on the initial value of `j` because any value in [0; 10] has an execution satisfying the assertion (consider, for instance the executions where the random choice [0; 1] always returns 0). However, a backward sufficient condition analysis would compute the set of initial states such that *all* executions necessarily satisfy the assertion. It will infer the condition: $\mathtt{j} \in [0; 5]$ as, even if [0; 1] always evaluate to 1 in the loop, `j <= 105` holds nevertheless. Necessary and sufficient conditions thus differ in the presence of non-determinism.

Sufficient conditions have many applications, including: contract inference, counter-example generation, optimizing compilation, verification driven by temporal properties, etc. Contract inference [2] consists in inferring sufficient conditions at a function entry that ensure that its execution is error-free. Similarly, counter-example generation [3] infers initial states that guarantee that the program goes wrong. Safety checks hoisting, a form of compiler optimization, consists in replacing a set of checks in a code portion (such as a loop or method) with a single check at the code entry. For instance, array bound check hoisting (as done in [4]) infers sufficient conditions under which all array accesses in a method are correct, and then inserts a dynamic test that branches to an optimized, check-free version of the method if the condition holds but reverts to the original method if it does not. A final example use is the verification of temporal properties of programs, such as CTL formulas, as done by Massé [5], where necessary post-conditions as well as necessary and sufficient preconditions are mixed to take into account the interplay of modalities ($\Box$ and $\Diamond$).

Formal methods Determining the conditions under which a program is correct corresponds to inferring Dijkstra's weakest liberal preconditions [6]. The support for non-deterministic expressions, which was not present in Dijkstra's original presentation, was later added by Morris [7]. Weakest preconditions, and more generally predicate transformers, form the base of current deductive methods (see [8] for a recent introduction). They rely on the use of theorem provers or proof assistants and, ultimately, require help from the user to provide predicates and proof hints. Weakest preconditions also appear in model-checking in the form of modal operators [9]. Many instances of model-checking are based on an exhaustive search in the state-space, represented in extension or symbolically, while recent instances build on the improvement in solvers (such as SAT modulo theory [10]) to handle infinite-state spaces. Refinement methods, such as counter-example-guided abstract refinement [11], also merge model-checking with weakest preconditions computed by solvers. In this article, we seek to solve the sufficient condition inference problem using *abstract interpretation* instead. The benefits are: full automation, support for infinite-state systems, parametrization by a choice of domain-aware semantic abstractions and dedicated algorithms, and independence from a generic solver, with the promise to scale up to large programs.

Abstract interpretation Abstract interpretation [12] is a very general theory of the approximation of program semantics. It stems from the fact that the simplest inference problems on programs are undecidable (or, at least, grow too quickly in cost with the size of the state space to be of any use), and so, approximations are required to achieve scalable and yet fully automatic analyses. It has been applied with some success to the automatic generation of invariants on industrial applications and led to commercial tools, such as Astrée [13]. Its principle is to replace the computations on state sets (so-called concrete semantics) with computations in computer-representable abstractions that, for the sake of efficiency, only represent a selected subset of program properties and ignore others. Numeric abstract domains, which reason on the numerical properties of variables, are widely used and much effort has been spent designing domains adapted to selected properties or achieving a selected cost versus precision trade-off. The two most popular domains are: the interval domain (introduced by Cousot and Cousot in [14]) that infers variable bounds, and the polyhedra domain (introduced by Cousot and Halbwachs in [15]) that infers affine inequalities on variables. A more recent example is the octagon domain [16], which infers unit binary inequalities and thus achieves a balance between intervals and polyhedra in terms of cost and precision. Classic abstract domains enjoy abstract operators for forward and backward analyses, but their backward operators are geared towards the inference of necessary conditions and not the inference of sufficient ones. Moreover, when an exact result cannot be computed (due to the limited expressiveness of the domain or the impracticability of computing a precise solution),