Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Semantics-based generation of verification conditions via program specialization ☆

E. De Angelis [a,c,*], F. Fioravanti [a,c,*], A. Pettorossi [b,c,*], M. Proietti [c,*]

[a] DEC, University "G. d'Annunzio" of Chieti-Pescara, Viale Pindaro 42, 65127 Pescara, Italy
[b] DICII, University of Rome Tor Vergata, Via del Politecnico 1, 00133 Roma, Italy
[c] CNR-IASI, Via dei Taurini 19, 00185 Roma, Italy

ARTICLE INFO

Article history:
Received 31 December 2015
Received in revised form 14 August 2016
Accepted 7 November 2016
Available online xxxx

Keywords:
Horn clauses
Program verification
Program specialization
Semantics of programming languages
Software model checking

ABSTRACT

We present a method for automatically generating verification conditions for a class of imperative programs and safety properties. Our method is parametric with respect to the semantics of the imperative programming language, as it generates the verification conditions by specializing, using unfold/fold transformation rules, a Horn clause interpreter that encodes that semantics.
We define a multi-step operational semantics for a fragment of the C language and compare the verification conditions obtained by using this semantics with those obtained by using a more traditional small-step semantics. The flexibility of the approach is further demonstrated by showing that it is possible to easily take into account alternative operational semantics definitions for modeling additional language features. We have proved that the verification condition generation takes a number of transformation steps that is linear with respect to the size of the imperative program to be verified. Also the size of the verification conditions is linear with respect to the size of the imperative program. Besides the theoretical computational complexity analysis, we also provide an experimental evaluation of the method by generating verification conditions using the multi-step and the small-step semantics for a few hundreds of programs taken from various publicly available benchmarks, and by checking the satisfiability of these verification conditions by using state-of-the-art Horn clause solvers. These experiments show that automated verification of programs from a formal definition of the operational semantics is indeed feasible in practice.

1. Introduction

A well-established technique for the verification of program correctness relies on the generation of suitable *verification conditions* (VCs, for short) starting from the program code [2,11,29]. Verification conditions are logical formulas whose satisfiability implies program correctness, and the satisfiability check can be performed, if at all possible (because, in general, the problem of verifying program correctness is undecidable), by using special purpose theorem provers or *Satisfiability*

☆ This paper is an extended, improved version of [12].

* Corresponding authors.
E-mail addresses: emanuele.deangelis@unich.it (E. De Angelis), fabio.fioravanti@unich.it (F. Fioravanti), adp@iasi.cnr.it (A. Pettorossi), proietti@iasi.cnr.it (M. Proietti).

http://dx.doi.org/10.1016/j.scico.2016.11.002
0167-6423/

Modulo Theories (SMT) solvers [4,16]. Recently, *constrained Horn clauses* have been proposed as a common encoding format for software verification problems, thus facilitating the interoperability of different software verifiers, and efficient solvers have been made available for checking the satisfiability of Horn-based verification conditions [4,10,16,28,32]. The notion of a constrained Horn clause we use in this paper is basically equivalent to the notion of a *Constraint Logic Programming* (CLP) clause [33]. The choice of either terminology depends on the context of use. Constraints are assumed to be formulas of any first order theory.

Typically, verification conditions are automatically generated, starting from the programs to be verified, by using *verification condition generators*. These generators are special purpose software components that implement algorithms for handling the syntax and the semantics of both the programming language in which programs are written and the class of properties to be verified. A VC generator takes as input a program written in a given programming language, and a property of that program to be verified, and by applying axiomatic rules à la Floyd–Hoare, it produces as output a set of verification conditions.

Having built a VC generator for a given programming language, to build a new VC generator for programs written in an extension of that language, or a different programming language, or even for programs written in the same language syntax but with a different language semantics (for instance, the big-step semantics, instead of the small-step semantics [52]) requires the design and the implementation of a new, ad hoc VC generator.

In this paper we present a method for generating verification conditions which is based on a CLP encoding of the operational semantics of the programming language and on the CLP program specialization technique which uses the unfold/fold transformation rules.

The use of CLP program specialization for analyzing programs is not novel. Peralta et al. [49] have used it for analyzing simple imperative programs and Albert et al. [1,27] for analyzing Java bytecode. In a previous work of ours [11] VCs are generated from a small-step semantics, for verifying imperative programs using iterated specialization. Here we extend and further develop the VC generation technique based on CLP specialization, and we demonstrate its generality and flexibility by showing that suitable customizations of the CLP specialization strategy are able to effectively deal with a multi-step semantics and several variants thereof. By the term *multi-step semantics*, which is folklore, we mean a hybrid between small-step semantics and big-step semantics, where: (i) the execution of each command, different from a function call, is formalized as a one-step transition from a state to the next one, and (ii) the execution of a function call is formalized as a sequence of one-step transitions from the state where the function is called to the one where the function evaluation terminates. We also show the scalability of our technique for VC generation through both a theoretical complexity analysis and the results we have achieved by using our implementation. Finally, we show, in an empirical way, that our specialization strategy returns VCs which are of high quality, in the sense that these VCs can effectively be handled by state-of-the-art solvers for checking the satisfiability of Horn clause verification conditions [4,16,28,32]. Actually, some solvers perform better on VCs generated by specialization, than on VCs generated by *ad hoc* algorithms.

Our verification method can be described as follows. Given an imperative program P and a property φ to be proved, we construct a CLP program I, which defines a nullary predicate `unsafe` such that P satisfies the property φ if and only if the atom `unsafe` is not derivable from I. The construction of the CLP program I depends on the following parameters: (i) the imperative program P, (ii) the operational semantics of the imperative language in which P is written, (iii) the property φ to be proved, and (iv) the logic that is used for specifying φ (in this case, the reachability of an unsafe state, that is, a state where φ does not hold).

The verification conditions are obtained by specializing program I with respect to its parameters. This specialization process is performed by applying semantics-preserving unfold/fold transformation rules [17]. The application of these rules is guided by a strategy particularly designed for verification condition generation, called *the VCG strategy*. Thus, the correctness of the verification conditions follows directly from the correctness of the unfold/fold transformation rules that are applied during program specialization.

When we perform the specialization of the CLP program I, we get the effect of removing from I the overhead due to the level of interpretation which is present in I because of the clauses defining the operational semantics of the imperative programming language. This specialization, called *the removal of the interpreter* in this paper, realizes the *first Futamura projection*, which is a well-known operation in the program specialization literature [35]. Indeed, by the first Futamura projection we have that the specialization of an interpreter written in a language L (CLP, in our case) with respect to a source program (written in C, in our case) has the effect of compiling the source program into L. The removal of the interpreter drastically simplifies program I by getting rid of the complex terms (including lists) that encode the commands of the imperative language and their operational semantics. Then, the simplified program, derived after the removal of the interpreter, is handled by using special purpose automatic tools for Horn clauses with linear integer arithmetic constraints [4,16,28,32].

In our approach, similarly to what is done in other papers [6,44,48], we use a formal representation of the operational semantics of the language in which the imperative programs are written, as an explicit parameter of the verification problem. One of the most significant advantages of this technique is that it enables us to design widely applicable VC generators for programs written in different programming languages, and for different operational semantics of languages with the same syntax, by making small modifications only.

Please cite this article in press as: E. De Angelis et al., Semantics-based generation of verification conditions via program specialization, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2016.11.002