Journal of Logical and Algebraic Methods in Programming 99 (2018) 97–113

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Instrumenting a weakest precondition calculus for counterexample generation [☆]

Sylvain Dailler [a,b], David Hauzar [a,b,c], Claude Marché [a,b,*], Yannick Moy [c]

[a] Inria, Université Paris-Saclay, F-91120 Palaiseau, France
[b] LRI, CNRS & Univ. Paris-Sud, F-91405 Orsay, France
[c] AdaCore, F-75009 Paris, France

A R T I C L E   I N F O

Article history:
Received 28 February 2017
Received in revised form 23 February 2018
Accepted 17 May 2018
Available online 29 May 2018

Keywords:
Deductive program verification
Weakest precondition calculus
Satisfiability modulo theories
Counterexamples

A B S T R A C T

A major issue in the activity of deductive program verification is to understand why automated provers fail to discharge a proof obligation. To help the user understand the problem and decide what needs to be fixed in the code or the specification, it is essential to provide means to investigate such a failure. We present our approach for the design and the implementation of *counterexample generation*, exhibiting values for the variables of the program where a given part of the specification fails to be validated. To produce a counterexample, we exploit the ability of SMT solvers to propose, when a proof of a formula is not found, a *counter-model*. Turning such a counter-model into a counterexample for the initial program is not trivial because of the many transformations leading from a particular piece of code and its specification to a set of proof goals given to external provers.

1. Introduction

Deductive program verification is an activity that aims at checking that a given program respects a given functional behavior. In this context, the expected behavior must be expressed formally by logical assertions, i.e. preconditions and postconditions, forming a *contract*. Deductive program verification proceeds by generating, from both the code and the formal specification, a set of logic formulas called *verification conditions* (VCs), typically via a *Weakest Precondition Calculus* [1] (WP for short). If one proves all generated VCs, then the program is guaranteed to satisfy its specification. In recent program verification environments like Dafny [2], OpenJML [3] and Why3 [4], VCs are proved using automated theorem provers, in particular those of the *Satisfiability Modulo Theories* (SMT) family such as Alt-Ergo [5], CVC4 [6] and Z3 [7]. These theorem provers are used as black-boxes that, given a VC, may produce three kinds of results:

1. The prover answers something meaning "yes, the VC is valid".
2. The prover answers anything else, meaning "I don't know", in other words the prover is not able to prove the VC for some reason.
3. The prover runs for too long a time (seemingly infinitely) or runs out of memory.

[☆] Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) of the French National Research Agency.
[*] Corresponding author at: LRI, CNRS & Univ. Paris-Sud, F-91405 Orsay, France.
E-mail address: Claude.Marche@inria.fr (C. Marché).

https://doi.org/10.1016/j.jlamp.2018.05.003
2352-2208/© 2018 Elsevier Inc. All rights reserved.

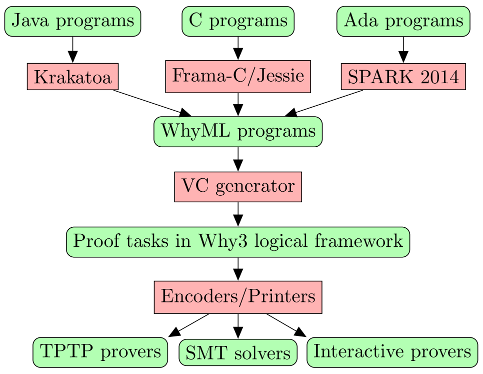

Fig. 1. Why3 ecosystem.

The case where the prover runs for too long a time is handled in practice by setting a given time limit, so that the prover process is killed when exceeding this limit. The cases 2 and 3 are the same from the user's perspective: the VC is not proved. Note that we do not distinguish a case where the prover would answer "no it is not valid", because the VCs typically involve undecidable logic features (e.g. non-linear integer arithmetic, first-order quantification) so provers are in practice incomplete: there is no way for them to be sure that a given VC is not provable.

A major issue in the activity of deductive verification is thus understanding the reasons for a proof failure. There are various reasons why it may fail:

1. The property to prove is indeed invalid: the code is not correct with respect to the given specification.
2. The property is in fact valid, but is not proved, again for two possible reasons:
   - The prover is not able to obtain a proof (in the given time and memory limits): this is the incompleteness of the proof search.
   - The proof might need extra intermediate annotations, such as loop invariants, or more complete contracts of the subprograms.

For the user to be able to fix the code or the specification of their program, it is essential to understand into which of these two cases any undischarged VC falls. The solution we propose in this paper is to generate *counterexamples*, giving values for the variables of the program, demonstrating a particular case where a given annotation may not hold. To produce a counterexample, we exploit an additional feature of SMT solvers: the ability to propose, when a proof of a formula is not found, a *counter-model*, exhibiting an interpretation of the free variables where the formula cannot be proved true. Turning such a counter-model into a counterexample for the initial program is not a trivial task because of the many transformations that lead to a VC from a given piece of code and its specification. It is important to notice here that because of the sources of incompleteness mentioned above, this process can only produce *candidate* counterexamples, in the sense that they do not necessarily exhibit an error in the code or in the specification. Given a counterexample, the user must analyze it to decide if it corresponds to a true bug or to an incompleteness (such as an extra loop invariant required).

The work presented here was conducted in the context of the Why3 environment for deductive program verification (http://why3.lri.fr), providing the language WhyML for specification and programming [8]. WhyML is used as an intermediate language for verification of programs written in C, Java or Ada [9,10]. A schematic view of the Why3 ecosystem is shown in Fig. 1. The specification component of WhyML [11], used to write program annotations and background theories, is an extension of first-order logic. The specification part of the language serves as a common format for theorem proving problems, *proof tasks* in Why3's jargon. Why3 generates proof tasks from user lemmas and annotated programs, using a weakest-precondition calculus, then dispatches them to multiple provers. Other deductive verification environments are structured in a similar way, for example the Boogie intermediate language [12] serves as an intermediate language for Spec#, VCC or Dafny, implements its own variant of a WP calculus, and dispatches the VCs to the SMT solver Z3. The approach we present here for counterexample generation thus applies to other environments as well, the main issue being how to trace the counter-model proposed by SMT solvers back through the WP calculus and furthermore back through the encoding of front-end languages to the intermediate language.

Our approach for generating counterexamples for the SPARK 2014 front-end was originally presented at the SEFM conference in 2016 [13]. The current article investigates in a more generic manner the issues we had to solve, and details the technical solutions we designed. We also present a technique for handling arrays that is different from the previous version. The experimental evaluation is updated with results of the newer implementation, with the new handling of arrays but also a few implementation bugs fixed. Also, it does not only make use of the CVC4 solver but also Z3. In Section 2, we present the design of an *instrumentation* of the WP calculus, aiming at keeping track of source code data inside the generated formulas. The *transformations* of proof goals, which need to be performed before sending the VCs to external provers, are instrumented as well. In Section 3, we focus on the additional instrumentation work that was needed to handle compound