Electronic Communications of the EASST
Volume 72 (2015)

Proceedings of the
15th International Workshop on
Automated Verification of Critical Systems (AVoCS 2015)

Studying Verification Conditions for Imperative Programs

Cláudio Belo Lourenço and Si-Mohamed Lamraoui and Shin Nakajima and Jorge Sousa Pinto

15 pages

Guest Editors: Gudmund Grov, Andrew Ireland
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Studying Verification Conditions for Imperative Programs

Cláudio Belo Lourenço[2] **and Si-Mohamed Lamraoui**[1] **and Shin Nakajima**[1] **and Jorge Sousa Pinto**[2]

National Institute of Informatics, Japan[1]
HASLab/INESC TEC & Universidade do Minho, Portugal[2]

Abstract:

Program verification tools use verification condition generators to produce logical formulas whose validity implies that the program is correct with respect to its specification. Different tools produce different conditions, and the underlying algorithms have not been properly exposed or explored so far. In this paper we consider a simple imperative programming language, extended with assume and assert statements, to present different ways of generating verification conditions. We study the approaches with experimental results originated by verification conditions generated from the intermediate representation of LLVM.

Keywords: software verification, LLVM, bounded verification, single-assignment

1 Introduction

Formal verification of programs is an approach to achieve required reliability levels of software-intensive systems. Given a correctness criterion, expressed as a set of *assertions*, we generate a formula in a suitable logic that encodes all potential execution paths together with the assertions (cf. [AFPS11]). Such a formula, known as a verification condition (VC), is then sent to a proof tool. If the proof succeeds, then we can conclude that the program under analysis satisfies the given correctness criterion. If it does not succeed, we hopefully extract as much information as possible to debug the program and/or the specification.

In the early days the reasoning methods relied on manual proof (cf. [LSS87]). Introducing some automation then became one of the major topics in the subject matter [FLL+02]. With the advancement of Boolean satisfiability (SAT) methods [MMZ+01] and the successful introduction of bounded model checking (BMC) [CKY03], program verification methods also adopted the SAT [CKL04] and the satisfiability modulo theories (SMT) [DM06] approaches to automated proof. If the VC refers to a formula in a decidable sub-fragment of a first-order theory, it is automatically discharged.

Whether with manual or with automatic proofs, the VC encoding is crucial for the efficiency and precision of the program verification method. Precision here refers to how much detailed information is obtained for debugging if the VC is not satisfied. Indeed, many different methods for generating VCs are used by existing program verification tools [CKL04, AMP08, MFS12, BCD+06, FP13]. In spite of this large diversity, systematic analyses of such VCs are lacking so far. Users may experience that a certain tool, adapting a certain encoding of VCs, is efficient for particular classes of programs, but not so well suited to other types of programs.

We have developed a tool infrastructure for systematic analyses of various VCs. Although the algorithm employed to generate VCs is different for each method, many components in the tool are common, such as the parser, the intermediate representation, and the interface to the backend solver. The algorithm to generate different VCs can be *plugged* into the framework. Since we can focus on a specific generation algorithm, comparing VCs can be transparent. We used the LLVM compiler infrastructure [LA04] as *host*, and developed our framework on top of it.

The first contribution of this paper is the presentation of different algorithms that generate VCs using different methods. Although the algorithms are standard in software verification, as far as we know, they have never been systematically presented – they are normally formalized using different notations and formalisms, or they are not formalized at all. With a common formalization we can highlight the differences on the generated VCs and reason about them.

This paper also reports empirical results from a study we have conducted with several VC generation algorithms implemented in our framework. The study reinforces the impression, from our use of various program verification tools mentioned above, that the choice of a method is not irrelevant. Furthermore, our study shows that choosing the most efficient VCs is not an easy task – no algorithm is better than the others in most situations – and that using proved assert statements as lemmas can improve solving time significantly.

This paper is organized as follows: Section 2 reviews the background. Section 3 describes several algorithms to generate VCs. In Section 4 after presenting the VCs comparison framework, we present our empirical results. Section 5 concludes the paper.

2 Background

Program verification tools are generally divided into two main categories: *deductive verification tools* [BCD$^+$06, FP13] and *model checking tools* [JM09] (a third important category of tools is that of abstract interpretation-based program analyzers, which fall outside the scope of our paper since they are not based on VC generation). While the former allows for a rich and expressive system of annotations through the use of *first-order logic*, the latter only allows simple quantifier-free properties to be used. In deductive verification, specifications are normally given as *contracts* (such as *preconditions* and *postconditions* of subprograms), as well as *loop invariants*, which are in general very difficult to generate automatically in a way that allows correctness to be proved. Although loop invariants are mandatory for the verification to proceed, in general user intervention is required in order to find them. In model checking, the verification process can proceed in a fully automatic way by automatically inserting properties (namely safety checks, such as overflow, division by zero, array out-of-bounds checks, etc.), or else, by inserting simple properties through the use of *assume* and *assert* statements. An assume statement is used to indicate that a property should be considered to hold at a given point of the program, and assert statements are used to check if a property is true at a given location.

The price to pay for automation in model checking is the lack of scalability due to *state-space explosion* [JM09]. To address this problem, two different approaches have been proposed along the years, which increase scalability either by sacrificing *completeness* (possibly reporting false alarms) in the case of *abstraction techniques* [JM09], or else by sacrificing *soundness* (considering only bounded executions) in the case of *bounded model checking of soft-*

```
assume(n >= 0);
pp := 0; p := 1; i := 2;
if(n == 0) r := pp else r := p;
while(i <= n){ assert(pp <= p);
  r  := p + pp;
  pp := p;
  p  := r;
  i  := i + 1;
}
```

```
assume(n0 >= 0);
pp0 := 0; p0 := 1; i0 := 2;
if(n0 == 0) r1 := pp0 else r2 := p0;
r3 := (n0 == 0)?r1:r2;
if(i0 <= n0){ assert(pp0 <= p0);
  r4  := p0 + pp0;
  pp1 := p0;
  p1  := r4;
  i1  := i0 + 1;
  assume(!(i1 <= n0)); }
r5  := (i0 <= n0)? r4: r3;
pp2 := (i0 <= n0)?pp1:pp0;
p2  := (i0 <= n0)? p1: p0;
i2  := (i0 <= n0)? i1: i0;
```

Figure 1: Fibonacci function (left) and its conversion to SSA after unwinding loop once (right)

ware [CKL04, AMP08, MFS12]. The latter technique generates potentially very large VCs, since it is based on loop unfolding, and for this reason the choice of a VC generation method may be particularly relevant.

In spite of the fundamental differences between deductive verification and BMC, the techniques also share many similarities. In both cases, state-of-the-art tools rely on the conversion of standard programs to an intermediate single-assignment form, from which a specific algorithm, called a verification condition generator (VCGen), generates VCs in the form of logical formulas. These VCs must then be discharged, that is, they must be sent to a solver for validity checking. If they are valid, the program is correct with respect to its specification.

In this paper, we will focus on single-assignment programs with no loops. Note, however, that we are not limiting our approach – programs with loops can be transformed into programs free of loops that can be verified instead of the original. In BMC tools (e.g. CBMC [CKL04]), loops are unwound k times (where k is given by the user or fixed/inferred by the tool) and an assume statement is inserted to ignore executions requiring further iterations. An alternative that generates sound VCs as opposed to BMC, is described in [BL05] and implemented in Boogie [BCD$^+$06]. Loops are replaced by a series of statements that simulates an arbitrary iteration of the loop. Here, loop invariants are required to restrict the verification to feasible iterations. In this paper we will focus exclusively on the BMC approach, because it generates larger VCs, resulting in more interesting formulas for our evaluation.

The basic principle of single-assignment (SA) forms is that once a variable has been used, it cannot be assigned. The most popular SA form, Static Single-Assignment (SSA) [CFR$^+$91], has been part of compiler pipelines for decades, and more recently it has been used in software verification tools. In this form each variable in a program can be assigned at most once. To synchronize variables assigned in branching blocks, *Phi-functions* (often replaced by *conditional expressions*) are used to capture the correct values at the merging point. An alternative approach, known as Dynamic Single-Assignment (DSA) form (e.g. [BL05, CFP12]), allows variables to be assigned more than once, as long as it is in different execution paths.

The example shown in Figure 1 (left) calculates the *n-th Fibonacci* number. On the right we show the result of unwinding the loop once and converting it to SSA form, following the BMC approach. The first assume statement is used as a precondition to specify that the verification

$$
\begin{aligned}
\mathsf{VC}^{\mathsf{se}} :\ & \Gamma^{*} \times \mathbf{Comm} \to \Gamma^{*} \times \Gamma^{*} \\
\mathsf{VC}^{\mathsf{se}}(\Phi, \mathbf{skip}) &= (\Phi, \emptyset) \\
\mathsf{VC}^{\mathsf{se}}(\Phi, x := e) &= (\{\phi \wedge x = e \mid \phi \in \Phi\}, \emptyset) \\
\mathsf{VC}^{\mathsf{se}}(\Phi, \mathbf{assume}\,\theta) &= (\{\phi \wedge \theta \mid \phi \in \Phi\}, \emptyset) \\
\mathsf{VC}^{\mathsf{se}}(\Phi, S_1 ; S_2) &= (\Phi_2, V_1 \cup V_2) \\
\text{where } (\Phi_1, V_1) &= \mathsf{VC}^{\mathsf{se}}(\Phi, S_1) \\
(\Phi_2, V_2) &= \mathsf{VC}^{\mathsf{se}}(\Phi_1, S_2) \\
\mathsf{VC}^{\mathsf{se}}(\Phi, \mathbf{if}\, b\, \mathbf{then}\, S^t\, \mathbf{else}\, S^f) &= (\Phi^t \cup \Phi^f, V^t \cup V^f) \\
\text{where } (\Phi^t, V^t) &= \mathsf{VC}^{\mathsf{se}}(\{\phi \wedge b \mid \phi \in \Phi\}, S^t) \\
(\Phi^f, V^f) &= \mathsf{VC}^{\mathsf{se}}(\{\phi \wedge \neg b \mid \phi \in \Phi\}, S^f) \\
\mathsf{VC}^{\mathsf{se}}(\Phi, \mathbf{assert}\,\theta) &= (\Phi, \{\phi \to \theta \mid \phi \in \Phi\})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{VC}^{\mathsf{sp}} :\ & \Gamma \times \mathbf{Comm} \to \Gamma \times \Gamma^{*} \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{skip}) &= (\top, \emptyset) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, x := e) &= (x = e, \emptyset) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{assume}\,\theta) &= (\theta, \emptyset) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, S_1 ; S_2) &= (F_1 \wedge F_2, V_1 \cup V_2) \\
\text{where } (F_1, V_1) &= \mathsf{VC}^{\mathsf{sp}}(\phi, S_1) \\
(F_2, V_2) &= \mathsf{VC}^{\mathsf{sp}}(\phi \wedge F_1, S_2) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{if}\, b\, \mathbf{then}\, S^t\, \mathbf{else}\, S^f) &= ((b \wedge F^t) \vee (\neg b \wedge F^f), V^t \cup V^f) \\
\text{where } (F^t, V^t) &= \mathsf{VC}^{\mathsf{sp}}(\phi \wedge b, S^t) \\
(F^f, V^f) &= \mathsf{VC}^{\mathsf{sp}}(\phi \wedge \neg b, S^f) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{assert}\,\theta) &= (\top, \{\phi \to \theta\})
\end{aligned}
$$

Figure 2: SE (left) and SP (right) auxiliary functions

procedure should only consider executions where n is greater than or equal to 0 (to perform modular verification, the same property must be checked with an assert statement at each calling point). The assert statement is used to check a basic property that in the recurrence relation of the Fibonacci sequence corresponds to $F_{n-2} \leq F_{n-1}$. The second assume statement is introduced when the loop is unwound; its role is to cause executions requiring more iterations to be ignored.

3 Verification Condition Generators

This section presents different VCGens for SA programs whose loops and function calls have been previously removed as explained in the previous section. The presented VCGens are sound and can be used interchangeably. We will be considering an elementary SA language of branching programs with integer type expressions, whose commands are as follows:

$$\mathbf{Comm} \ni C \ ::= \ \mathbf{skip} \mid \mathbf{assume}\,\theta \mid C ; C \mid x := e \mid \mathbf{if}\, b\, \mathbf{then}\, C\, \mathbf{else}\, C \mid \mathbf{assert}\,\theta$$

with x ranging over a set of SA variables, e over integer expressions, and b and θ over Boolean expressions (θ may range instead over some richer assertion language, for instance in the context of deductive verification). We do not fix the language of integer and Boolean expressions, but we do restrict it to be a language that can be encoded in the chosen backend solver. The VCGens will take as input a command $C \in \mathbf{Comm}$, and will return a set of VCs in the form of logical formulas (in what follows we denote those as Γ) whose validity implies that the program is correct. The program in Figure 1 will be used as a running example to show differences in the generated VCs. For more details about theoretical aspects, such as soundness results, refer to [BFS15].

3.1 Symbolic Execution

Symbolic execution (e.g. [JMNS12]) is the simplest way of generating VCs. The approach consists of generating a VC for every single execution path that reaches an assert statement, such that the validity of each VC ensures that executions going through the encoded path satisfy the assert statement. To prove that a program is fully correct, a VC for each possible execution path and each assert statement must be generated and then discharged.

Definition 1 (SE VCGen) Given an SA program C, its *Symbolic Execution VCs* are given by the set V where $(E,V) = \mathsf{VC}^{\mathsf{se}}(\emptyset, C)$, and $\mathsf{VC}^{\mathsf{se}}$ is the function given in Figure 2 (left).

Function $\mathsf{VC}^{\mathsf{se}}$ in Figure 2 receives an extra parameter in addition to the input program and also returns an extra formula in addition to the set of VCs. The extra parameter corresponds to the encoding of the part of the program that precedes the current statement. The extra formula is composed by the conjunction of the received formulas, with the encoding of the current statement. When an assert statement is encountered, the extra parameter contains the encoding of all possible execution paths that reach this assert statement. A VC will be generated for each of these paths. The generated SE VCs for our running example are as follows:

$$VC1 : n_0 \geq 0 \wedge pp_0 = 0 \wedge p_0 = 1 \wedge i_0 = 2 \wedge n_0 = 0 \wedge r_1 = pp_0 \wedge r_3 = ((n_0 = 0)?r_1 : r_2) \wedge i_0 \leq n_0 \rightarrow pp_0 \leq p_0$$
$$VC2 : n_0 \geq 0 \wedge pp_0 = 0 \wedge p_0 = 1 \wedge i_0 = 2 \wedge \neg(n_0 = 0) \wedge r_2 = p_0 \wedge r_3 = ((n_0 = 0)?r_1 : r_2) \wedge i_0 \leq n_0 \rightarrow pp_0 \leq p_0$$

The first VC corresponds to executions going through the *then* branch in the first *if* statement, and the other to executions going through the *else* branch. Even though the generated VCs are only two and they are relatively small in this case, it is important to note that the number of paths is, in the worst case, exponential with respect to the size of the program, and so is the number of generated VCs [CFP12].

3.2 Efficient Strongest Postcondition

The *weakest precondition* (WP) and *strongest postcondition* (SP) *predicate transformers* introduced by Dijkstra [Dij76] for his *guarded commands language* may also be used to generate VCs. These techniques predate symbolic execution, and in their original form do not require the program to be transformed into an SA form. For non-SA programs the technique produces VCs whose size is, in the worse case, exponential with respect to the size of the program. Flanagan and Saxe showed that when the technique was applied to SA programs, the size of the generated VCs was, in the worst case, quadratic [FS01]. In this paper we focus exclusively on SP VCs for SA programs. Due to lack of space we omit the discussion of WP which generates similar VCs.

Definition 2 (SP VCGen) Given an SA program C, its *Strongest Postcondition VCs* are given by V where $(E,V) = \mathsf{VC}^{\mathsf{sp}}(\top, C)$, and $\mathsf{VC}^{\mathsf{sp}}$ is the function given in Figure 2 (right).

Definition 2 differs slightly from the standard definition of SP, because the asserted properties are not being introduced into the context – we leave this discussion for Section 3.5. The function $\mathsf{VC}^{\mathsf{sp}}$, similarly to $\mathsf{VC}^{\mathsf{se}}$ (Figure 2), receives an auxiliary parameter and returns an extra formula containing exclusively the encoding of the present statement. The part of the program that has already been encoded is now propagated by the *if* and *sequence* rules. A disjunction is introduced to encode each branching statement. The generated SP VC for our running example is as follows:

$$n_0 \geq 0 \wedge pp_0 = 0 \wedge p_0 = 1 \wedge i_0 = 2 \wedge ((n_0 = 0 \wedge r_1 = pp_0) \vee (\neg(n_0 = 0) \wedge r_2 = p_0)) \wedge r_3 = (n_0 = 0?r_1 : r_2) \wedge i_0 \leq n_0 \rightarrow pp_0 \leq p_0$$

3.3 Conditional Normal Form

In the context of BMC, after unwinding loops, inlining function calls and converting the code to an SA form, VCs are generated by performing a series of transformations on the code. In particular the following rules are applied to conditional branches:

$$
\begin{aligned}
\mathsf{VC}^{\mathsf{cnf}} &: \Gamma \times \Gamma^* \times \mathbf{Comm} \to \Gamma^* \times \Gamma^* \\
\mathsf{VC}^{\mathsf{cnf}}(\pi, C, \mathbf{skip}) &= (\emptyset, \emptyset) \\
\mathsf{VC}^{\mathsf{cnf}}(\pi, C, x := e) &= (\{\pi \to x = e\}, \emptyset) \\
\mathsf{VC}^{\mathsf{cnf}}(\pi, C, \mathbf{assume}\,\theta) &= (\{\pi \to \theta\}, \emptyset) \\
\mathsf{VC}^{\mathsf{cnf}}(\pi, C, S_1\,;S_2) &= (C_1 \cup C_2, P_1 \cup P_2) \\
\text{where } (C_1, P_1) &= \mathsf{VC}^{\mathsf{cnf}}(\pi, C, S_1) \\
(C_2, P_2) &= \mathsf{VC}^{\mathsf{cnf}}(\pi, C \cup C_1, S_2) \\
\mathsf{VC}^{\mathsf{cnf}}(\pi, C, \mathbf{if}\, b\, \mathbf{then}\, S^t\, \mathbf{else}\, S^f) &= (C^t \cup C^f, P^t \cup P^f) \\
\text{where } (C^t, P^t) &= \mathsf{VC}^{\mathsf{cnf}}(\pi \wedge b, C, S^t) \\
(C^f, P^f) &= \mathsf{VC}^{\mathsf{cnf}}(\pi \wedge \neg b, C, S^f) \\
\mathsf{VC}^{\mathsf{cnf}}(\pi, C, \mathbf{assert}\,\theta) &= (\emptyset, \{\pi \to \theta\})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{VC}^{\mathsf{scnf}} &: \Gamma \times \Gamma^* \times \mathbf{Comm} \to \Gamma^* \times \Gamma^* \\
\mathsf{VC}^{\mathsf{scnf}}(\pi, C, \mathbf{skip}) &= (\emptyset, \emptyset) \\
\mathsf{VC}^{\mathsf{scnf}}(\pi, C, x := e) &= (\{x = e\}, \emptyset) \\
\mathsf{VC}^{\mathsf{scnf}}(\pi, C, \mathbf{assume}\,\theta) &= (\{\pi \to \theta\}, \emptyset) \\
\mathsf{VC}^{\mathsf{scnf}}(\pi, C, S_1\,;S_2) &= (C_1 \cup C_2, P_1 \cup P_2) \\
\text{where } (C_1, P_1) &= \mathsf{VC}^{\mathsf{scnf}}(\pi, C, S_1) \\
(C_2, P_2) &= \mathsf{VC}^{\mathsf{scnf}}(\pi, C \cup C_1, S_2) \\
\mathsf{VC}^{\mathsf{scnf}}(\pi, C, \mathbf{if}\, b\, \mathbf{then}\, S^t\, \mathbf{else}\, S^f) &= (C^t \cup C^f, P^t \cup P^f) \\
\text{where } (C^t, P^t) &= \mathsf{VC}^{\mathsf{scnf}}(\pi \wedge b, C, S^t) \\
(C^f, P^f) &= \mathsf{VC}^{\mathsf{scnf}}(\pi \wedge \neg b, C, S^f) \\
\mathsf{VC}^{\mathsf{scnf}}(\pi, C, \mathbf{assert}\,\theta) &= (\emptyset, \{\pi \to \theta\})
\end{aligned}
$$

Figure 3: CondNF (left) and SCondNF (right) auxiliary functions

1. $\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \Longrightarrow \mathbf{if}\ b\ \mathbf{then}\ C_1\,;\ \mathbf{if}\ \neg b\ \mathbf{then}\ C_2$
2. $\mathbf{if}\ b\ \mathbf{then}\ \{C_1\,; C_2\} \Longrightarrow \mathbf{if}\ b\ \mathbf{then}\ C_1\,;\ \mathbf{if}\ b\ \mathbf{then}\ C_2$
3. $\mathbf{if}\ b_1\ \mathbf{then}\ \mathbf{if}\ b_2\ \mathbf{then}\ C \Longrightarrow \mathbf{if}\ b_1 \wedge b_2\ \mathbf{then}\ C$

Note that these transformations would not be sound in a standard imperative program (a program not in SA form). For instance, in the case 1 above, the execution of C_1 could modify some variable used in b causing both branches of the conditional branch to be executed. Since in the SA form, once a variable has been used cannot be assigned, we have a sound transformation.

After the above transformations the program consists of a sequence of conditional statements of the form **if** b **then** C, with C an atomic (assignment, assume, or assert) statement. This is the so-called *conditional normal form* (CondNF) of an SA program. From this form, it is easy to extract a VC. In this paper, instead of applying each transformation individually, we present an algorithm that internally captures the above transformations with a single pass through the program and generates the corresponding VC.

Definition 3 (CondNF VCGen) Given an SA program C, its *Conditional Normal Form VC* is the formula $\bigwedge E \to \bigwedge P$, where $(E, P) = \mathsf{VC}^{\mathsf{cnf}}(\top, \emptyset, C)$, and $\mathsf{VC}^{\mathsf{cnf}}$ is the function given in Figure 3 (left).

The $\mathsf{VC}^{\mathsf{cnf}}$ takes two parameters in addition to the program. The first contains the path condition for the present statement to be reached. The second, which is not being used in this definition, contains the encoding of the program up to the present statement. The set E contains the operational encoding of the program (assignment statements) and assume statements, and the set P contains the asserted properties. Regardless of the number of execution paths and the number of properties, only one VC, whose size is in the worse case quadratic, is generated [CFP12].

The use of a single global context implies that the semantics of assume statements is different from the previous techniques: whereas before an assume statement dispensed executions from having to pass subsequent assert statements, in the CondNF the use of a global context means that an assume statement now dispenses executions from passing any of the assert statements they meet. An alternative approach consists of generating multiple VCs with partial contexts – each assert property is only implied by the relevant part of the program. This can be done by generating a VC each time an assert statement is encountered – recall that the second parameter in the auxiliary function $\mathsf{VC}^{\mathsf{cnf}}$ contains the encoding of the program up to the present statement.

Definition 4 (PCondNF VCGen) Given an SA program C, its *Partial context Conditional Normal Form VCs* are given by V, where $(E,V) = \mathsf{VC}^{\mathsf{pcnf}}(\top,\emptyset,C)$, and $\mathsf{VC}^{\mathsf{pcnf}}$ is the function in which $\mathsf{VC}^{\mathsf{pcnf}}(\pi,C,\textbf{assert}\,\theta) = (\emptyset, \bigwedge C \rightarrow \pi \rightarrow \theta)$ and $\mathsf{VC}^{\mathsf{pcnf}}(\pi,C,S) = \mathsf{VC}^{\mathsf{cnf}}(\pi,C,S)$, for $S \in \textbf{Comm} \setminus \textbf{assert}$ and $\mathsf{VC}^{\mathsf{cnf}}$ given in Figure 3 (left).

For our running example, $\mathsf{VC}^{\mathsf{cnf}}$ produces the following sets of formulas which are then used by CondNF VCGen (Definition 3) to generate a VC:

$$\begin{aligned}
E = \{&\top \rightarrow n_0 \geq 0, \top \rightarrow pp_0 = 0, \top \rightarrow p_0 = 1, \top \rightarrow i_0 = 2, n_0 = 0 \rightarrow r_1 = pp_0, \neg(n_0 = 0) \rightarrow r_2 = p_0, \top \rightarrow r_3 = ((n_0 = 0)?r_1 : r_2),\\
&i_0 \leq n_0 \rightarrow r_4 = p_0 + pp_0, i_0 \leq n_0 \rightarrow pp_1 = p_0, i_0 \leq n_0 \rightarrow p_1 = r_4, i_0 \leq n_0 \rightarrow i_1 = i_0 + 1, i_0 \leq n_0 \rightarrow \neg(i_1 \leq n_0),\\
&\top \rightarrow r_5 = (i_0 \leq n_0)?r_4 : r_3, \top \rightarrow pp_2 = (i_0 \leq n_0)?pp_1 : pp_0, \top \rightarrow p_2 = (i_0 \leq n_0)?p_1 : p_0, \top \rightarrow i_2 = (i_0 \leq n_0)?i_1 : i_0\}\\
P = \{&i_0 \leq n_0 \rightarrow pp_0 \leq p_0\}
\end{aligned}$$

The VC generated by PCondNF VCGen (Definition 4) is as follows:

$$\begin{aligned}
&((\top \rightarrow n_0 \geq 0) \wedge (\top \rightarrow pp_0 = 0) \wedge (\top \rightarrow p_0 = 1) \wedge (\top \rightarrow i_0 = 2) \wedge (n_0 = 0 \rightarrow r_1 = pp_0) \wedge (\neg(n_0 = 0) \rightarrow r_2 = p_0)\\
&\wedge (\top \rightarrow r_3 = ((n_0 = 0)?r_1 : r_2))) \rightarrow (i_0 \leq n_0 \rightarrow pp_0 \leq p_0)
\end{aligned}$$

In the case of CondNF, as opposed to PCondNF, the whole program is used as a context even though only part of it is required. In both cases, every statement is encoded with its path condition, resulting in larger formulas than those coming from the predicate transformer VCGens.

3.4 Simplified Conditional Normal Form

The previous VCGens generate sound VCs from programs in either SSA or DSA form. There exists however an additional point of interest in the use of SSA form. Since a new variable is introduced after each branch statement to synchronize both branches, we can encode assignment statements without considering their path conditions. This approach omits part of the control flow, but this is not a problem because the relevant information is propagated through Phi-functions. Although the transformation does not produce operationally equivalent programs, it captures the necessary information to generate sound VCs. The next definition enhances the CondNF VCGen (Definition 3) in this way.

Definition 5 (SCondNF VCGen) Given an SSA program C, its *Simplified Conditional Normal Form VC* is the formula $\bigwedge E \rightarrow \bigwedge P$, where $(E,P) = \mathsf{VC}^{\mathsf{scnf}}(\top,\emptyset,C)$, and $\mathsf{VC}^{\mathsf{scnf}}$ is the function given in Figure 3 (right).

Note that, in the auxiliary function $\mathsf{VC}^{\mathsf{scnf}}$, only the rule that encodes assignments is different from $\mathsf{VC}^{\mathsf{cnf}}$. As for CondNF, the previous definition can be adapted to generate multiple VCs with partial contexts, resulting in a similar algorithm as the one used by CBMC [CKL04].

Definition 6 (PSCondNF VCGen) Given an SSA program C, its *Partial context Simplified Conditional Normal Form VCs* are given by V, where $(E,V) = \mathsf{VC}^{\mathsf{pscnf}}(\top,\emptyset,C)$, and $\mathsf{VC}^{\mathsf{pscnf}}$

is the function in which $\mathsf{VC}^{\mathsf{pscnf}}(\pi, C, \textbf{assert}\,\theta) = (\emptyset, \bigwedge C \to \pi \to \theta)$ and $\mathsf{VC}^{\mathsf{pscnf}}(\pi, C, S) = \mathsf{VC}^{\mathsf{scnf}}(\pi, C, S)$, for $S \in \textbf{Comm} \setminus \textbf{assert}$ and $\mathsf{VC}^{\mathsf{scnf}}$ given in Figure 3 (right).

Function $\mathsf{VC}^{\mathsf{scnf}}$ produces the set of formulas as follows to be used by SCondNF (Definition 5)

$$\begin{aligned}E = \{&n_0 \geq 0, pp_0 = 0, p_0 = 1, i_0 = 2, r_1 = pp_0, r_2 = p_0, r_3 = ((n_0 = 0)?r_1 : r_2), r_4 = p_0 + pp_0, pp_1 = p_0, p_1 = r_4, i_1 = i_0 + 1,\\ &i_0 \leq n_0 \to \neg(i_1 \leq n_0), r_5 = (i_0 \leq n_0)?r_4 : r_3, pp_2 = (i_0 \leq n_0)?pp_1 : pp_0, p_2 = (i_0 \leq n_0)?p_1 : p_0, i_2 = (i_0 \leq n_0)?i_1 : i_0\}\\ P = \{&i_0 \leq n_0 \to pp_0 \leq p_0\}\end{aligned}$$

and the PSCondNF VCGen (Definition 6) produces the following VC:

$$((n_0 \geq 0) \wedge (pp_0 = 0) \wedge (p_0 = 1) \wedge (i_0 = 2) \wedge (r_1 = pp_0) \wedge (r_2 = p_0) \wedge (r_3 = ((n_0 = 0)?r_1 : r_2))) \to (i_0 \leq n_0 \to pp_0 \leq p_0)$$

The observations we made about the number and size of the VCs generated by CondNF and PCondNF are also applied to SCondNF and PSCondNF. Note however that the size of the formulas is slightly smaller, because the path conditions are not used for assignment statements.

3.5 Assert Statements as Lemmas

We have presented different ways of generating verification conditions, but we have not actually discussed how to solve them. It is important to note that we are assuming that one VC is checked at a time. Whenever a VC is false, one can stop the verification process because a violation has been found. If all VCs are valid, one can say that the program is correct.

It is interesting to point out that assert statements, once proved, can be used as lemmas to prove the subsequent properties. One way of doing it is by inserting the *proved* assert statements' properties in the context of the subsequent assert statements. In this case the VCs must be generated and solved in the same order as they appear in the code, so that all lemmas are proved before they are referred to. The previous VCGen algorithms can be modified to reproduce this behavior as in the following definition.

Definition 7 (VCGens with assert statements as lemmas) Given a $VCGen \in \{SE, SP, PCondNF, PSCondNF\}$ and its corresponding auxiliary function $\mathsf{VC} \in \{\mathsf{VC}^{\mathsf{se}}, \mathsf{VC}^{\mathsf{sp}}, \mathsf{VC}^{\mathsf{pcnf}}, \mathsf{VC}^{\mathsf{pscnf}}\}$, we write $VCGen_l$ (resp. VC_l) to indicate that assert statements are used as lemmas. In this case, $\mathsf{VC}_l(S) = \mathsf{VC}(..., S)$ for $S \in \textbf{Comm} \setminus \textbf{assert}$ and $\mathsf{VC}_l(..., \textbf{assert}\,\theta)$ is given as follows:

1. $\mathsf{VC}_l^{\mathsf{se}}(\Phi, \textbf{assert}\,\theta) = (\{\phi \wedge \theta \mid \phi \in \Phi\}, \{\phi \to \theta \mid \phi \in \Phi\})$
2. $\mathsf{VC}_l^{\mathsf{sp}}(\phi, \textbf{assert}\,\theta) = (\theta, \{\phi \to \theta\})$
3. $\mathsf{VC}_l^{\mathsf{pcnf}}(\pi, C, \textbf{assert}\,\theta) = (\{\pi \to \theta\}, \{\bigwedge C \to \pi \to \theta\})$
4. $\mathsf{VC}_l^{\mathsf{pscnf}}(\pi, C, \textbf{assert}\,\theta) = (\{\pi \to \theta\}, \{\bigwedge C \to \pi \to \theta\})$

These lemmas cannot be used by $\mathsf{VC}^{\mathsf{cnf}}$ and $\mathsf{VC}^{\mathsf{scnf}}$ because only one formula is generated with the whole encoding of the program and the assert statements – if we added the asserted properties to the global context, the generated VC would be trivially (and wrongly) discharged.

For our running example this modification would not produce any effect, because only one assert statement is present. Note however that if the loop statement was unwound twice, two assert statements would exist, one from each iteration. The assert statement from the first iteration could be used as lemma for the second iteration.

```
x1 = Δ1;
assert(θ1);
...
xn = Δn;
```

CondNF VC: $x_1 = \Delta_1 \wedge \ldots \wedge x_n = \Delta_n \rightarrow \theta_1$

SP VC: $\{x_1 = \Delta_1 \rightarrow \theta_1\}$

```
x1 = Δ1;
assert(θ1);
...
xn = Δn;
assert(θn);
```

CondNF VC: $x_1 = \Delta_1 \wedge \ldots \wedge x_n = \Delta_n \rightarrow \theta_1 \wedge \theta_2$

SP VCs: $\{x_1 = \Delta_1 \rightarrow \theta_1, \ldots, x_1 = \Delta_1 \wedge \ldots \wedge x_n = \Delta_n \rightarrow \theta_n\}$

Figure 4: Left: example whose CondNF VC's size is linear and SP is constant; Right: example whose CondNF VC's size is linear and SP is quadratic

3.6 A Glance Over the Differences

Both the efficient predicate transformers and CondNF VCGens are able to curb the exponential growth of the global VCs' size that is characteristic of SE and of the original predicate transformer methods. Although both algorithms produce VCs that cannot grow faster than quadratically in the size n of the program, one cannot be said to be better than the other: think for instance of the program shown in Figure 4 (left), consisting only of a sequence of assignment statements and a single assert statement at the beginning – the CondNF VC has linear size because it contains a global context, and the SP VC has constant size; now, imagine a similar program as shown in Figure 4 (right), where an assert statement follows each assignment – CondNF VC still has linear size, but the SP VCs have quadratic size, since there are n conditions, of size 1 to $\Theta(n)$. In these examples PCondNF would generate similar VCs to those generated by the SP VCGen. For different VCs to be generated, examples with conditional branches (as was the case of our running example) must be considered.

The size of CondNF and PCondNF verification conditions can be reduced by taking advantage of the SSA form. This is achieved by the SCondNF and PSCondNF VCGens, where the flow of the program is captured by the Phi-functions only. Note however, that in this case, the solver might have to evaluate all assignments because they are not guarded by the path condition.

Lastly we have shown that VCs generated with partial contexts can be easily modified to make use of valid assert statements as lemmas for subsequent properties. Note however that it is not guaranteed that the lemmas will simplify the proofs. Therefore if this approach is applied blindly it may produce bigger encodings without adding useful information to the context.

4 Evaluation

Let us now present the results of an empirical study to compare the VCGens in terms of *solving time*. We focus exclusively on the solving time, disregarding other factors such as the time for parsing, applying the transformations referred in Section 2, generating VCs, or even encoding the formulas in the backend solver, since it is clearly the solving process that dominates the growth of the verification time. Moreover, in the present paper we base our study on VCs generated by unwinding loops (VCs generated in a deductive verification setting may well be more complex). In what follows, when we write that a VCGen is more efficient than another, we mean that the VCs generated by the first are faster to solve than those generated by the second.

We have created a VCGen comparison framework based on components from the SNIPER

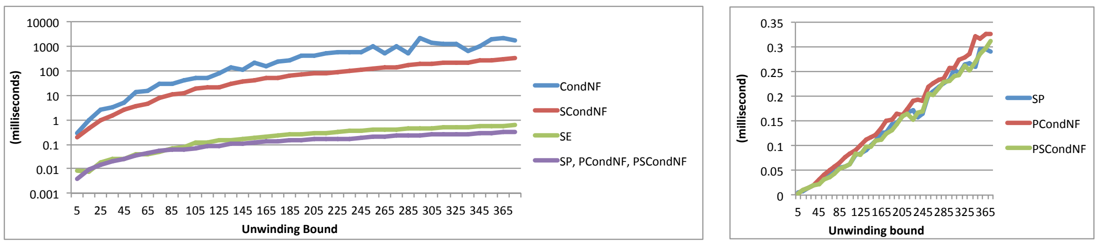


Figure 5: Time to solve VCs generated from Fibonacci running example

tool set [LN14]. New VCGens can easily be added to the framework, and we have adapted all the previous definitions for the LLVM intermediate representation (IR) [LA04]. We have chosen LLVM IR firstly because it is already in SSA form, and secondly because different languages, such as C, Ada or Objective-C, can be compiled into LLVM IR, which means that a single framework can be used for multiple input languages (this paper focuses on a subset of the C programming language). Yices [DM06], version 1.2, is used as backend SMT solver; following [AMP08] we use linear integer arithmetic to check the validity of the generated VCs. The user can choose whether asserts are added to the context with a command line flag.

The first part of this section shows the evaluation of the VCGens with the Fibonacci function. We compare how the solving time grows as the unwinding bound is increased. Even though the example is very simple, it already illustrates some properties of the VCGens. We then use more complex programs taken from different benchmarks previously used to validate verification tools (e.g. [GCNR08, GR09]). First we expose the difficulties of selecting the most efficient VCGen, then we motivate the use of assert statements as lemmas. Due to the inefficiency of the SE VCGen we only consider it for Fibonacci, where the exponential growth is not manifest. All tests were performed on a 1.7GHz MacBook Air with 4GB of RAM and OS X 10.10.

4.1 The Fibonacci Running Example

Let us start by analysing our running example as the unwinding bound increases. The solving time for each VCGen is shown in Figure 5. For this particular example, adding assert statements as lemmas does not produce significant differences in the solving time (neither positively nor negatively). In the log-scale chart on the left we have merged SP, PCondNF, and PSCondNF together, since their solving time is almost the same. From the chart on the right we can see that SP and PSCondNF are slightly faster than PCondNF (on average 13% faster). The use of path conditions in the PCondNF to guard the assignment statements does not improve the solving time – instead it just increases the formula size and the solving time.

It can be observed from the results that the SE solving time is not growing exponentially. This is justified by the fact that the number of paths that reach each assert statement is always two: only one path results from the code obtained from unwinding the loop; this code consists of nested *if* statements, where the assert statement is always in the path in which all precedent *if* conditions are true. Therefore, the most relevant difference when comparing with SP, PCondNF and PSCondNF is that two VCs are generated for each assert statement instead of one. Another interesting lesson we can draw from this chart is that irrelevant code, which is present in CondNF

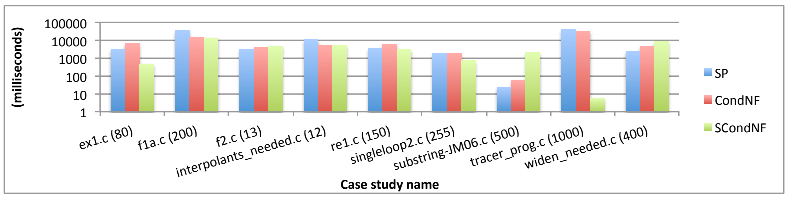


Figure 6: Case studies used to validate multiple verification tools (e.g. [GCNR08, JMNS12])

and SCondNF due to the fact that the entire program is encoded (including code after the assert statements), may heavily degrade the solving time, even if only one formula has to be solved.

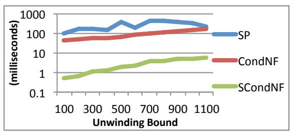
4.2 Electing the Most Efficient VCs

The first question that arises when evaluating different algorithms with a common goal is: "which one is the most efficient?". One should address this question by means of an empirical study, running the different algorithms with different benchmark programs. Ideally, one particular algorithm would stand out with the best behavior for all benchmarks, but our study, described below, shows that this is not the case.

Let us first consider a benchmark[1] whose case studies have been previously used to test and validate a wide variety of software verification tools (e.g. [GCNR08, JMNS12]). In this benchmark, assume statements are exclusively used as preconditions (placed at the beginning of the program) and assert statements are exclusively used as postconditions (placed at the end of the program). Therefore, it only makes sense to compare SP, CondNF and SCondNF: all others (apart from SE) will generate equivalent VCs. All case studies have loops, which allows us to adjust the complexity of the input program.

Different test target programs were selected to illustrate how the most efficient VCs can change from one target benchmark program to another. Results are shown in Figure 6: the x-axis shows different programs and the number of times they were unwound; the y-axis shows the solving time (in logarithmic scale) for the generated VCs. The most consistent pattern is that CondNF is never the best option, but it is also seldom the worst. All the others vary from case to case.

```
assume(x==0 && y>=0);
while (x < 100) {
  y = y + 1;
  x = x + 1;}
assert(!(y+x<100));
```



```
assume(x>=0 && y==0);
while (x < 100) {
  y = y + 1;
  x = x + 1;}
assert(!(y>100));
```

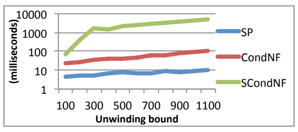


Figure 7: Two variants of tracer_prog_d.c and solving time as loop unwinding bound is increased

Let us now focus on a particular benchmark target program, to show how subtle modifications in the code or specification can change drastically the efficiency of a particular VCGen. Figure 7 shows two similar variants (1) and (2) of a particular case study, and the required time to solve the generated VCs, as the loop unwinding bound is increased. While SCondNF is the most

[1] Available from http://map.uniroma2.it/smc/simp/

efficient VCGen for (1) and SP is the less efficient, for (2) SP is the most efficient and SCondNF is the less efficient. Note also that the solving time is longer in (2). The main difference between both variants is that (1) iterates exactly 100 times, while (2) iterates between 0 and 100 times depending on the value of the variable x. In (1), 100 assignments are always reached, but in (2) the number of assignments that are reached depends on the value of a variable whose value is greater than or equal to zero. In both cases, only dead code results from unwinding the loop more than 100 times. We imagine that the inefficiency of PCondNF in (2) is justified by the amount of unguarded expressions. Surely the VCs generated from (1) also have unguarded expressions, but they can be easily solved if the SMT solver applies some simplification strategy, such as constant propagation. In this case, having a condition guarding assignment statements only increases the size of the expressions that have to be simplified. In the chart from (2) note also how the SCondNF solving time increases drastically when loops are unwound more than 100 times: it is precisely at this point that some dead code is added.

It can be argued that it is not useful to unwind the loop more than 100 times, and that this is an artificial problem. Therefore we considered a variant where we replaced the value 100 in the loop condition and assert statement, by an undetermined variable. Naturally the solving time for both sets of VCs increased. However, PSCondNF is no longer the most efficient solver for (1). Instead, the solving time is now very similar for the different sets of VCs. In the variant (2), SP still produces the most efficient VCs and PSCondNF the most inefficient.

After looking at this case study one could intuitively think that SP is the most efficient VCGen, since it performs better in the presence of indeterminately-valued variables, but we remark that this is not always true. For instance, the first target program shown in Figure 6 has a structure that depends on indeterminate variables, but SCondNF still generates the most efficient VCs.

4.3 Assert Statements as Lemmas

The previous examples are already complex enough, but they follow a specific pattern for using assume and assert statements as preconditions and postconditions respectively, which does not allow us to compare the algorithms of Section 3.5. An additional benchmark[2] we consider in our evaluation has been used to test Invgen [GR09], an invariant generator tool. It contains iterative programs with relatively complex data flow, and some of the examples contain multiple assert statements spread throughout the program. We present here a representative selection of the results obtained with this benchmark. We do not consider the CondNF and SCondNF algorithms, which in general perform worse than PCondNF and PSCondNF.

The chart in Figure 8 shows the required solving time for each set of VCs generated from a particular program using different VCGens. Due to space constraints, instead of discussing each case study separately let us simply categorize the differences in terms of the usage of assert statements. The first three contain multiple assert statements throughout the program. The next three contain assert statements inside the loop(s), resulting in multiple assert statements as the loops are unwound. The last three contain assert statements at the end of the program only.

It can be observed from the chart that in many cases the time required for solving the generated VCs decreases when assert statements are used as lemmas. In other cases, the lemmas do not

[2] Available from http://www.tcs.tifr.res.in/~agupta/invgen/

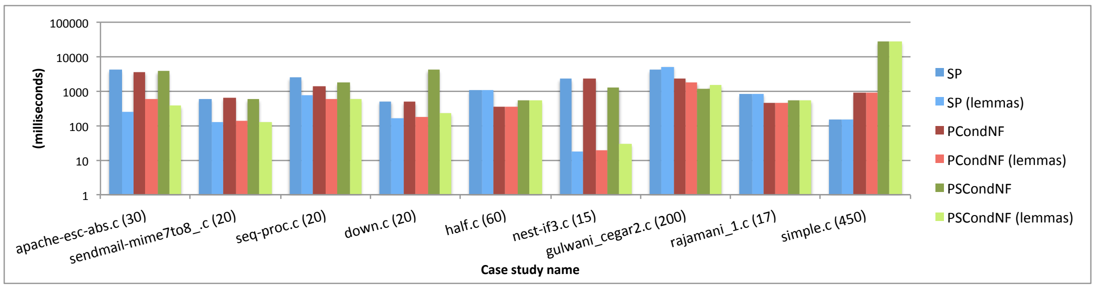


Figure 8: Case studies used to validate Invgen

improve the solving time, but they also do not degrade it considerably. The last two case studies have a single assert statement at the end, therefore the VC that is produced is the same, whether assert statements are used as lemmas or not.

5 Conclusion and Future Work

As far as we know, this is the first time that the VCGens are presented in a systematic way – they are normally formalized using different languages and concepts, or they are not formalized at all. We have used a simple imperative programming language to expose the algorithms in a very concise and clear way, which allowed us to highlight the differences between them. On the other hand, our implementation of the algorithms is based on the LLVM intermediate representation, used by real-world applications. Its native SSA form allowed us to implement and compare all the presented algorithms in a straightforward way.

We remark that a crude comparison of VCGen algorithms in isolation, as we have carried out, may be unfair to some of them, since their performance may depend on the combination with other techniques. For instance the SE VCGen is impractical due to its inefficiency, but it can be seen as the basis of tools like TRACER [JMNS12], which employs an interpolation technique, based on weakest preconditions and unsatisfiable cores, for detecting infeasible paths and avoiding exponential path enumeration.

Although we have opted to use forward propagation for the sake of uniformity, the SP VC-Gen produces VCs similar to those produced by the Boogie tool [BCD$^+$06], which is based on efficient weakest preconditions. CondNF VCs are described for instance in [CKL04, BCD$^+$06, AMP08], and the PSCondNF VCGen is our interpretation of the method used by the CBMC tool, which includes partial contexts and the SSA-specific optimizations described in Section 3.4.

The empirical study reveals that it is not possible to select a single most efficient VCGen – it varies from case to case. In general, we have seen that solving a large VC, generated by CondNF and SCondNF, is slower than solving multiple smaller VCs generated by PCondNF and PSCondNF respectively. However, in exceptional cases, solving a single VC is slightly faster than solving multiple smaller ones. Moreover, it is sometimes useful to guard the assignments, but not always: sometimes the guards create larger formulas, increasing the solving time.

What seems to contribute decisively to decrease the solving time, is to use proved assert statements as lemmas. This never seems to increase the solving time considerably, and in some cases reduces it considerably. Therefore, adopting these variants is recommended in general.

We also used faulty programs for our evaluation, but the results mostly followed what has been mentioned before. Of course, if an assert statement at the beginning of the program is violated, the VCs that contain only partial contexts will rapidly fail – on the other hand, VCs with a global context will be extremely inefficient. If there is a violated assert in the final part of the program, VCs with partial contexts sometimes perform worse than VCs with global contexts.

Our evaluation results show that the choice of VCGen algorithm is not irrelevant, but no single 'best' algorithm exists. This suggests that it is a good idea for verification tools to provide a number of different VCGens, leaving to the end user the decision of which one to use (in the same way that some tools allow for the use of various solvers). This way, a default VCGen may be used, but if it does not scale as desired, an alternative VCGen can be chosen. We note however that switching VCGens may not always be possible or easily achievable, since the VCGen algorithm may be deeply integrated into the tool in a way that makes it difficult to decouple.

As future work, it is important to pursue efforts to categorize the kinds of programs for which each VCGen performs better. In particular, it would be interesting to have a wrapper algorithm or heuristic to choose the best VCGen for each situation. We also believe that it would be interesting to make a similar study using DSA instead of SSA – this is in fact a limitation of our framework, because LLVM does not support DSA representation. Also, we do not consider in this paper multiple backend solvers; it would be equally interesting to observe the efficiency of different VCs in different solvers. Finally, considering the simplification of VCs before they are sent to the backend solver – applying constant propagation or rewriting of expressions (as done by CBMC) – can produce new insight for a future evaluation.

Acknowledgements: This work is partially financed by the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project UID/EEA/50014/2013 and by the NII International Internship Program.

Bibliography

[AFPS11] J. B. Almeida, M. J. Frade, J. S. Pinto, S. M. Sousa. *Rigorous software development: an introduction to program verification*. Springer-Verlag, 2011.

[AMP08] A. Armando, J. Mantovani, L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Int. J. STTT*, 11(1):69–83, 2008.

[BFS15] C. B. Lourenço, M. J. Frade, and J. S. Pinto. A Tutorial on Verification Conditions Using Single-Assignment Form. *(unpublished draft available from http://haslab.uminho.pt/jsp), University of Minho*, 2015.

[BCD+06] M. Barnett, B. Chang, R. DeLine, B. Jacobs, K. Leino. Boogie: a modular reusable verifier for object-oriented programs. In *Proc. FMCO2006*, pp. 364–387, 2006.

[BL05] M. Barnett, K. Leino. Weakest-precondition of unstructured programs. In *Proc. PASTE2005*, pp. 82–87, 2005.

[CFP12] D. Cruz, M. J. Frade, J. S. Pinto. Verification conditions for single-assignment programs. In *Proc. SAC2012*, pp. 1264–1270, 2012.

[CFR$^+$91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Int. J. TOPLAS*, 13(4):451–490, 1991.

[CKL04] E. Clarke, D. Kroening, F. Lerda. A Tool for Checking ANSI-C Programs. In *Proc. TACAS2004*, pp. 168–176, 2004.

[CKY03] E. Clarke, D. Kroening, K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proc. DAC2003*, pp. 368–371, 2003.

[Dij76] E. W. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.

[DM06] B. Dutertre, L. de Moura. The Yices SMT solver. Tech. rep., SRI Int., 2006.

[FLL$^+$02] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended static checking for Java. In *PLDI2002*, pp. 234–245, 2002.

[FP13] J.C. Filliâtre, A. Paskevich. Why3 – where programs meet provers. In *Proc. ESOP2013*, pp. 125–128, 2013.

[FS01] C. Flanagan, J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proc. POPL2001*, pp. 193–205, 2001.

[GCNR08] B. S. Gulavani, S. Chakraborty, A. V. Nori, S. K. Rajamani. Automatically refining abstract interpretations. In *Proc. TACAS2008*, pp. 443–458, 2008.

[GR09] A. Gupta, A. Rybalchenko. InvGen: an efficient invariant generator. In *Proc. CAV2009*, pp. 634–640, 2009.

[JM09] R. Jhala, R. Majumdar. Software model checking. *ACM Computer Survey* 41(4):21:1–21:54, 2009.

[JMNS12] J. Jaffar, V. Murali, J. A. Navas, A. E. Santosa. TRACER: a symbolic execution tool for verification. In *Proc. CAV2012*, pp. 758–766, 2012.

[LA04] C. Lattner, V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *Proc. CGO2004*, pp. 75–86, 2004.

[LN14] S.M. Lamraoui, S. Nakajima. A formula-based approach for automatic fault localization of imperative programs. In *Proc. ICFEM2014*, pp. 251–266, 2014.

[LSS87] J. Loeckx, K. Sieber. *The Foundations of Program Verification, 2nd ed.* Wiley-Teubner, 1987.

[MFS12] F. Merz, S. Falke, C. Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In *Proc. VSTTE2012*, pp. 146–161, 2012.

[MMZ$^+$01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: engineering an efficient SAT solver. In *Proc. DAC2001*, pp. 530–535, 2001.