A Tutorial on Verification Conditions Using Single-Assignment Form

CLÁUDIO LOURENÇO
and
MARIA JOÃO FRADE
and
JORGE SOUSA PINTO

HASLab & INESC-TEC
Universidade do Minho

The use of an intermediate *single-assignment form* is central in the design of major modern program verification tools, from deductive verifiers like Boogie and Why3 to automated tools like the CBMC *bounded model checker*. In this tutorial paper we formalize the program verification technique that consists in the translation of branching code to such an intermediate single-assignment form, from which verification conditions are generated. Soundness and completeness results are established for the technique, and different methods for generating the verification conditions are studied and compared.

Categories and Subject Descriptors: []:

General Terms:

Additional Key Words and Phrases:

Contents

1. INTRODUCTION

In recent years program verification tools have attained a very interesting level of applicability. This is in particular true for programming languages used in the development of embedded (and often safety-critical) software, as exemplified by the SPARK programming language toolset (a commercial product) and by a plethora of tools for checking ANSI-C code. These tools can mostly be grouped in two categories, based on completely different principles, but which have both been progressing steadily to higher degrees of maturity, taking advantage of recent progresses in satisfiability solving technology.

On one side one finds tools based on *deductive* principles, inspired by the notions of *program logic*. These tools analyze the code to produce a set of *verification conditions*, first-order formulas that are then sent to a backend satisfiability solver. This approach is not automatic, in the sense that it relies on *program annotations* introduced by users in the code, corresponding to *loop invariants* and *contracts* for subprograms, which may include *preconditions*, *postconditions*, and *frame conditions*, among other elements.

On the other side one finds *software model checking* tools. These are fully automatic tools that apply the widely popular principles of model checking in the context of software. Since these tools work without user intervention (no annotations are expected in the code), scalability is a problem (recall the classic problem of *state space explosion* in model checking). The tools address this problem in one of two ways: either by introducing an *abstraction mechanism*, or by performing a *bounded* exploration of the state space. The former solution results in a sound but incomplete technique: no property violations are missed, but *false positives* (spurious counterexamples) may be present. The latter solution on the other hand is in its basic form unsound, since it will not detect property violations occurring deep in the structure of the execution (deeper than considered by the current bounded model being explored).

This paper is an introduction to the logical foundations underlying program verification tools, in particular deductive verifiers and bounded model checkers (abstraction techniques are left out of our discussion). It includes a self-contained treatment of the semantic foundations of verification condition generation, but great importance is given to a fundamental element that is usually not considered in texts on

program verification: the use of *single-assignment code* in intermediate forms for program verification. Single-assignment forms are used explicitly by bounded model checkers of software, and implicitly as part of the internal workflow of deductive verification tools, for reasons that have to do both with efficiency and with the advantages offered in the treatment of contracts (in technical terms, single-assignment forms solve what is usually known as the *adaptation* problem).

Our explicit use of single-assignment forms in this paper allows us to give a novel formal account of the workflow on which most deductive verification tools are based. But, very importantly, it also allows us to introduce verification condition generation in a much more intuitive way than is usually the case in program verification texts. In fact, single-assignment code is much closer to a logical reading than imperative programs in general, as will become clear in what follows.

The Program Verification Challenge. We will start by writing down the basic challenge of formal program verification in simple terms, in a way that can be explained to an experienced programmer with some knowledge of logic.

First of all, recall that many programming languages now contain an *assert* or *check* statement, which can be placed anywhere in a program, and carries a condition (Boolean expression) as argument, like:

$$\textbf{assert } x > 0$$

This statement can be used by compilers that feature *dynamic verification* capabilities: special code can be included in the executable program that will check, in runtime, that the condition is satisfied whenever the statement is executed. Once the program has been sufficiently validated and becomes ready to go into production, it can be compiled with the verification switch turned off, so that this special code will no longer be included. Dynamic verification has advantages over testing, in particular for the compositional validation of code using a methodology usually known as *design by contract*.

But dynamic verification is not the subject of this work. The program verification challenge is how to check the program *statically*, once and for all, by analyzing the code rather than executing it. We could formulate this challenge as follows:

> *You are given an imperative program containing assert statements. Compiler technology tools are readily available, and so is a theorem proving tool that supports reasoning with the data types of your programming language. How can you use the tools at your disposal to ensure that* every execution of the program *successfully passes all the assert statements contained in it?*

Many software engineers would be capable of solving this problem in some way or another, based on their understanding of the semantics of imperative programs. It is basically a matter of generating logical formulas from the code in such a way that if the formulas are valid (as checked by the theorem prover) then the code is correct (i.e. no assert is violated in any execution). Such formulas are usually known as *verification conditions* (VCs).

We will in fact work with a slightly richer notion of correctness, which includes the use of *assume* as a second command for specification purposes. When included

at the beginning of a program, an assume statement serves as a *precondition*, which is precisely an assertion that is assumed to hold when the program is executed. A precondition limits the set of executions that are of interest for the correctness of the program. For instance a program that computes the division of two positive integers x and y could have a precondition written as

$$\textbf{assume}\, x > 0 \wedge y > 0$$

and a postcondition written as

$$\textbf{assert}\, x = y * q + r \wedge r < y$$

The property expressed by the *assert* only needs to apply to executions that satisfy the precondition: only these executions have to be considered for verification. But on the other hand one is warned that the behavior of the program with non-positive numbers is unknown (not specified).

It is perhaps slightly less intuitive to understand the role of an *assume* statement in the middle of the code, but the principle is the same. Again it allows to eliminate concrete executions from being considered: any execution that fails to pass an $\textbf{assume}\,\theta$ statement, when it is met at an arbitrary point in the code, does not have to satisfy any *assert* commands that may subsequently be executed.

To express this more rigorously, let us say that the execution of a command $\textbf{assume}\,\theta$ or $\textbf{assert}\,\theta$ in the state s *succeeds* if s satisfies θ. A program will be correct if, in every execution, every assert command succeeds when all the assume commands *executed before it* have also succeeded.

Structure of this Paper. In this paper we first give, in Section 2, a tour of the different techniques that can be used to generate verification conditions for branching programs, based on the conversion to an intermediate single-assignment form: Symbolic Execution, Conditional Normal Form Transformation, Predicate Transformers, and the "efficent" version of the latter. We prove various correspondence results between techniques, and then present a novel unified view of the most efficient of these, in the form of a generalized Verification Condition Generator algorithm (VCGen).

Section 3 is an overview of program verification for While programs (extended with assume and assert commands) formulated in the standard framework consisting of a natural semantics, the corresponding interpretation of Hoare triples, and the inference system of Hoare logic. We extend this framework to allow us to formulate the generation of verification conditions, by introducing a notion of annotated program; a goal-directed inference system for triples containing annotated programs; and finally a VCGen algorithm. We then give appropriate soundness and completeness results for VC generation.

Section 4 establishes a bridge between the two previous sections: using the tools reviewed in Section 3, we prove formally that the verification technique that consists of first translating a program to dynamic SA form, and then generating efficient VCs from this form, is sound and complete. We do this by first defining rigorously our notion of SA program, then proposing a dedicated program logic for these programs and deriving one of the VCGens of Section 2 from this logic, and finally defining a general notion of translation of Hoare triples into SA form.

```
1 if (x<0) x = -x;
2 else if (y<0) y = -y;
3 if (c>0) c = c-1;
4 assert x >= 0;
```

Listing 1. A non-iterating program

We end our study in Section 5 considering how the ideas introduced here extend to programs with iteration, and to the rich behavioral specification languages employed by typical deductive verification tools. Finally, we explain another advantage of using intermediate SA forms for program verification, which is the fact that the reasoning becomes *adaptation-complete*.

2. VERIFICATION CONDITIONS FOR SINGLE-ASSIGNMENT BRANCHING PROGRAMS

In this section we will answer our initial question by writing algorithms for the generation of verification conditions for programs not containing iterating constructs (in other words, for branching programs), whose specification is given by arbitrarily placed *assert* and *assume* statements. We will review and compare a number of algorithms that have been proposed in different contexts.

Throughout the section we will use the SAVER tool, a Python script that parses[1] a program in SA form and is able to generate verification conditions using the techniques that will be described. SAVER uses the Z3 SMT solver (programmatically, through the z3py API) to attempt to discard the generated verification conditions, but also incorporates a pretty-printing mode that simply displays the VCs.

```
$ ./saver.py --help
usage: saver.py [-h] [--se] [--sp] [--splitsp] [--cnf] [--print] [infile]

Verification Conditions for IMP programs.

positional arguments:
  infile      filename containing IMP program

optional arguments:
  -h, --help  show this help message and exit
  --se        Symbolic Execution VCs
  --sp        Efficient Strongest Postcondition
  --splitsp   Split Efficient Strongest Postcondition
  --cnf       CNF-like VCs
  --print     Print generated VCs (no conversion to Z3)
```

Let us start by looking at a very simple imperative program, not containing any iterating code, no *assume* statements, and a single *assert* statement at the end. The fragment in Listing 1, simple as it is, exemplifies how conditional statements

[1]The SAVER parser is based on Jay Conrod's parser combinators,
http://www.jayconrod.com/posts/37/a-simple-interpreter-from-scratch-in-python-part-1

can be composed to complicate the branching structure of programs: the first conditional contains an additional conditional in one of its branches, so it has 3 different execution paths; and it is in turn composed sequentially with another conditional, which duplicates the number of execution paths to 6.

2.1 Symbolic Execution

One way, maybe the simplest, to approach the problem, is to generate a formula for each execution path, such that the validity of each such formula ensures that every execution going through that path is safe (i.e. does not violate any assert). Doing this for every execution path covers the entire space of executions of the program.

The program in Listing 1 contains a single assert statement at the end, which can be seen as a *postcondition* for the program. It should be fairly intuitive that in this case each verification condition will be an implicative formula, having as antecedent an encoding of the execution path, and as consequent the condition in the assert.

Consider for instance the execution path that is followed when x is negative and c is positive. In this case the assignments $x = -x$ and $c = c - 1$ are executed, and the execution path can be encoded by combining the conditions $x < 0$ and $c > 0$ with these assignments. The first problem we encounter is that a statement like $c = c - 1$ can obviously not be written simply as an equality (it is not a satisfiable formula!). One way to solve this (again, maybe the simplest way) is to use different logical variables for the same program variable, so that each logical variable will capture the value taken by the program variable in a given state of execution. The most natural choice is to use indexed variables, such that x_0 corresponds to the initial value of variable x, x_1 to the value taken by x after it has been assigned once, and so forth. If we follow this approach the execution path we have been considering can be encoded by the formula

$$x_0 < 0 \wedge x_1 = -x_0 \wedge c_0 > 0 \wedge c_1 = c_0 - 1$$

which allows us to write the following verification condition to be sent to the theorem prover:

$$x_0 < 0 \wedge x_1 = -x_0 \wedge c_0 > 0 \wedge c_1 = c_0 - 1 \rightarrow x_1 \geq 0$$

(note that the version of x to be used in the assert condition must be the last one introduced before the assert statement).

This explanation will hopefully have convinced the reader that it is straightforward to generate the formulas we seek. Let us now see how they can be generated *systematically* (so that the program can be verified automatically). It turns out that the best thing to do is to convert the program into a *single-assignment* (SA) form in which the logical variables used for versions of the same (program) variable become in fact different program variables. For instance, the program

```
x = y+1; x = x*10; assert x > 0
```

will be converted into

```
x1 = y0+1; x2 = x1*10; assert x2 > 0
```

Such a program is of course not operationally equivalent to the original, but the transformation is sound in the sense that if the latter program is correct then so is the initial one.

So what would our running example program look like in this form? It contains branching, which complicates the matters. Consider the second conditional:

```
if (c>0) c = c-1;
```

One could be tempted to convert it into single-assignment form as:

```
if (c0>0) c1 = c0-1;
```

This would be wrong, since there does not exist a single variable corresponding to the original c in the state immediately following the conditional. Consider for instance that this was followed by an assignment statement, say x = c. The following SA form would be wrong

```
if (c0>0) c1 = c0-1;
x1 = c1;
```

since in the case when the condition fails, the value of x1 should be set to c0, not c1. The usual way to solve this is to introduce an additional variable c2, which gets assigned in both branches, yielding

```
if (c0>0) {
  c1 = c0 - 1;
  c2 = c1;
}
else c2 = c0;
```

and there is now no doubt that a subsequent statement like x = c should be written in the SA form as x1 = c2. This may at first seem strange, since in the original program no assignment is performed in the else branch, but the use of c2 in both branches is required to allow the version variables in both branches to be synchronized. It is important to understand that from the point of view of the logical encoding, it is perfectly valid for the same variable to be assigned in different branches of the program, since these occurrences will appear in formulas corresponding to different execution paths. The single-assignment program resulting from applying this approach to our original example is shown in Listing 2.

Once a program has been converted to SA form, it is straightforward to generate VCs by considering each execution path individually: for each assert statement in the program, a different formula will be generated for each path that reaches the statement. Before writing formally how the VCs are calculated, we should mention the major problem of generating verification conditions in this way, which is that the number of execution paths (and consequently the number of VCs) is of course in the worst case exponential in the length of the program. To understand this, it suffices to consider a program consisting of a sequence of n conditional statements: since each such statement duplicates the number of execution paths (all incoming paths may be extended by going through both branches), the program has 2^n paths, with n linear in the length of the program. We will soon see that this exponential explosion is not an inherent feature of verification conditions, but of the path enumeration that is characteristic of symbolic execution.

```
1  if (x0<0) {
2    x1 = -x0;
3    x2 = x1;
4    y3 = y0;
5  }
6  else {
7    if (y0<0) {
8      y1 = -y0;
9      y2 = y1;
10   }
11   else y2 = y0;
12   x2 = x0;
13   y3 = y2;
14 }
15 if (c0>0) {
16   c1 = c0 - 1;
17   c2 = c1;
18 }
19 else c2 = c0;
20 assert x2 >= 0;
```

Listing 2. Example program in SA form

The box in page 10 shows how the SAVER tool can be used to both print and prove correctness of a program based on this technique. The box also shows the 6 verification conditions generated by the tool for the program of Listing 2.

Formal Definition. Let us formalize the technique by considering an elementary language of branching programs with integer-type expressions:

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

We will write a function that computes the set of VCs generated by symbolic execution.

In addition to the VCs themselves, the function computes the execution paths as it goes along. To be more precise it computes the formulas encoding these paths, as described previously, and accumulates them in the first parameter of the function, which contains the set of paths that reach the current statement. This set will be used to generate VCs when an assert is reached. A helper function explicitly computes the set of (formulas describing the) execution paths of a given program, again taking as first parameter a set of accumulated incoming paths to be used in the base cases.

NOTATION 1. *Given a set of formulas* Γ *and a formula* ϕ*, we will write* $\Gamma \wedge \phi$ *to denote the set of formulas* $\{\gamma \wedge \phi \mid \gamma \in \Gamma\}$*, and* $\Gamma \to \phi$ *to denote the set* $\{\gamma \to \phi \mid \gamma \in \Gamma\}$.

DEFINITION 2 SYMBOLIC EXECUTION VCS OF AN SA PROGRAM. *Given an SA program* C*, the set of its* symbolic execution VCs *is given as* $\mathsf{VC}^{\mathsf{se}}(\emptyset, C)$*, where the*

Static and Dynamic Single-assignment Forms

Translation into Static Single Assignment (SSA) form [Cytron et al. 1991] has been part of the standard compilation pipeline for decades now; in such a program each variable occurs at most once as the left-hand side of an assignment instruction. A construct called "Phi function" is used to synchronize versions of the same variable used in different execution paths. For instance the fragment

```
if (x>0) x = x+10;
else x = x+20;
```

could be translated as

```
if (x0>0) x1 = x0+10;
else x2 = x0+20;
x3 = Phi(x1,x2);
```

An alternative notion is that of *dynamic single assignment* (DSA) form [Vanbroekhoven et al. 2007], in which variables may occur in multiple assignments as long as they are not assigned more than once *in each execution*, which dispenses with the use of Phi functions. The above fragment could be written:

```
if (x0>0) { x1 = x0+10; x3 = x1 }
else { x2 = x0+20; x3 = x2 }
```

This is the form that we have been using in this section. It should however be noted that static single-assignment forms are also very popular in program verification, in particular in *software model checking*, where a transition relation is often extracted from code by first converting the latter into SSA – see for instance [Kroening 2009]. In *bounded model checking of software* [Clarke et al. 2004], loops are unfolded a given number of times, and the resulting branching code is then converted to an SSA form. These approaches employ C *conditional expressions* to encode Φ functions. Recall that the evaluation of the expression `b?e1:e2` proceeds by evaluating `b`, and then evaluating `e1` if `b` evaluates to true, and `e2` otherwise. Using such an expression it is possible to write the code above as

```
if (x0>0) x1 = x0+10;
else x2 = x0+20;
x3 = (x0 > 0) ? x1 : x2;
```

Observe that the logical value of the condition is necessarily preserved during execution of the conditional, since the variables occurring in it are not assigned. As such, the condition can be tested to check which branch has just been executed.

The example of Listing 1 could be written in SSA form with conditional expressions as follows:

```
if (x0<0) x1 = -x0;
else {
  if (y0<0) y1 = -y0;
  y2 = (y0 < 0) ? y1 : y0;
}
x2 = (x0 < 0) ? x1 : x0;
y3 = (x0 < 0) ? y0 : y2;
if (c0>0) c1 = c0 - 1;
c2 = (c0 > 0) ? c1: c0;
assert x2 >= 0;
```

Symbolic execution with SAVER

The SAVER tool calculates verification conditions by symbolic execution from a single-assignment program when invoked with the switch `--se`. They can be pretty-printed with the `--print` switch; without this switch the VCs will be sent to the solver for checking. The result of both invocations is shown below. The file `prog-sac.imp` contains the program of Listing 2.

```
$ ./saver.py --se  --print progs/prog-sac.imp
GENERATING SYMBOLIC EXECUTION VCs...
VC 1 :
(((((((x0<0 && x1==0-x0) && x2==x1) && y3==y0)
       && c0>0) && c1==c0-1) && c2==c1) ==> x2>=0)
VC 2 :
(((((((((!x0<0 && y0<0) && y1==0-y0) && y2==y1) && x2==x0) && y3==y2)
       && c0>0) && c1==c0-1) && c2==c1) ==> x2>=0)
VC 3 :
(((((((((!x0<0 && !y0<0) && y2==y0) && x2==x0) && y3==y2)
       && c0>0) && c1==c0-1) && c2==c1) ==> x2>=0)
VC 4 :
((((((x0<0 && x1==0-x0) && x2==x1) && y3==y0)
       && !c0>0) && c2==c0) ==> x2>=0)
VC 5 :
(((((((((!x0<0 && y0<0) && y1==0-y0) && y2==y1) && x2==x0) && y3==y2)
       && !c0>0) && c2==c0) ==> x2>=0)
VC 6 :
(((((((!x0<0 && !y0<0) && y2==y0) && x2==x0) && y3==y2)
       && !c0>0) && c2==c0) ==> x2>=0)


$ ./saver.py --se  progs/prog-sac.imp
GENERATING SYMBOLIC EXECUTION VCs...
VC 1 :
solving...
proved
VC 2 :
solving...
proved
VC 3 :
solving...
proved
VC 4 :
solving...
proved
VC 5 :
solving...
proved
VC 6 :
solving...
proved
```

function $\mathsf{VC}^{\mathsf{se}}$ *and its helper function* paths *are defined as follows:*

$$
\begin{aligned}
\mathsf{paths}(\Phi, \mathbf{skip}) &= \Phi \\
\mathsf{paths}(\Phi, x := e) &= \Phi \wedge x = e \\
\mathsf{paths}(\Phi, \mathbf{assume}\,\theta) &= \Phi \wedge \theta \\
\mathsf{paths}(\Phi, C_1 \,;\, C_2) &= \mathsf{paths}(\mathsf{paths}(\Phi, C_1), C_2) \\
\mathsf{paths}(\Phi, \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= \mathsf{paths}(\Phi \wedge b, C^t) \cup \mathsf{paths}(\Phi \wedge \neg b, C^f) \\
\mathsf{paths}(\Phi, \mathbf{assert}\,\theta) &= \Phi \wedge \theta
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{VC}^{\mathsf{se}}(\Phi, \mathbf{skip}) &= \emptyset \\
\mathsf{VC}^{\mathsf{se}}(\Phi, x := e) &= \emptyset \\
\mathsf{VC}^{\mathsf{se}}(\Phi, \mathbf{assume}\,\theta) &= \emptyset \\
\mathsf{VC}^{\mathsf{se}}(\Phi, C_1 \,;\, C_2) &= \mathsf{VC}^{\mathsf{se}}(\Phi, C_1) \cup \mathsf{VC}^{\mathsf{se}}(\mathsf{paths}(\Phi, C_1), C_2) \\
\mathsf{VC}^{\mathsf{se}}(\Phi, \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= \mathsf{VC}^{\mathsf{se}}(\Phi \wedge b, C^t) \cup \mathsf{VC}^{\mathsf{se}}(\Phi \wedge \neg b, C^f) \\
\mathsf{VC}^{\mathsf{se}}(\Phi, \mathbf{assert}\,\theta) &= \Phi \rightarrow \theta
\end{aligned}
$$

Observe how execution paths are extended with conditions when entering branches. In the case of a sequence of statements, the paths coming out of the first statement are fed into the accumulator to be extended with the traversal of the second statement.

We remark that the two functions could be merged into a single tupling one, returning a pair of sets of execution paths and verification conditions. This would in fact eliminate a source of inefficiency in the above definition, which in the case of sequential composition may compute paths with redundancy. We leave it to the reader to write this alternative definition with tupling.

2.2 Conditional Normal Form Transformation

In 2004 a technique called *Bounded Model Checking* (BMC) of software was proposed that avoids path enumeration and the consequent exponential explosion in the size of the verification conditions. The principle behind BMC is the bounded expansion of loops to produce a piece of branching code that can be converted to SA form. We will leave a more detailed discussion of this treatment of loops to a later section, and discuss now the technique used by BMC tools to encode iteration-free programs.

It is fascinating to observe how different solutions have been independently proposed for the problem of checking properties of programs; from our understanding, the encoding of programs used in bounded model checking of software was probably guided more by intuition than by formal semantics. What is remarkable is that it is both simple to understand (it is based on a series of program transformations, each of which is simple enough to be understood as "obviously sound"), and clever enough to avoid path enumeration and the consequent exponential explosion.

The variant of the technique that we will consider here starts by converting programs into a dynamic single-assignment form, as seen in Section 2.1. It then makes use of the following observations, valid for SA programs:

(1) The two branches of a conditional can be sequentialized. In other words, the following program transformation rule produces a program that is equivalent to the original:

$$\textbf{if}\ (\mathrm{b})\ \mathrm{C1}\ \textbf{else}\ \mathrm{C2} \implies \textbf{if}\ (\mathrm{b})\ \mathrm{C1};\ \textbf{if}\ (!\mathrm{b})\ \mathrm{C2}$$

In a normal program this would of course not be a sound transformation, since in the case that b evaluates to true, the execution of C1 may well modify the value of b, causing *both* branches of the conditional to be executed. But note that in a single-assignment program this is not possible, since the variables occurring in b will surely not be assigned in C1.

(2) For similar reasons, conditions can be distributed through the sequenced statements in the body of a conditional:

$$\textbf{if}\ (\mathrm{b})\ \{\mathrm{C1};\ \mathrm{C2}\} \implies \textbf{if}\ (\mathrm{b})\ \mathrm{C1};\ \textbf{if}\ (\mathrm{b})\ \mathrm{C2}$$

Since the value of b does not change with the execution of C1, the second conditional will be executed if and only if the first one was.

(3) Finally, nested conditionals can be simplified by joining conditions as follows:

$$\textbf{if}\ (\mathrm{b1})\ \textbf{if}\ (\mathrm{b2})\ \mathrm{C} \implies \textbf{if}\ (\mathrm{b1}\ \&\&\ \mathrm{b2})\ \mathrm{C}$$

```
1  // After first transformation step
2  if (x0<0) {
3    x1 = -x0;
4    x2 = x1;
5    y3 = y0;
6  }
7  if !(x0<0) {
8    if (y0<0) {
9      y1 = -y0;
10     y2 = y1;
11   }
12   if !(y0<0) y2 = y0;
13   x2 = x0;
14   y3 = y2;
15 }
16 if (c0>0) {
17   c1 = c0 - 1;
18   c2 = c1;
19 }
20 if !(c0>0) c2 = c0;
21 assert x2 >= 0;
22
23
24 // After second transformation step
25 if (x0<0) x1 = -x0;
26 if (x0<0) x2 = x1;
27 if (x0<0) y3 = y0;
28 if !(x0<0) if (y0<0) y1 = -y0;
29 if !(x0<0) if (y0<0) y2 = y1;
30 if !(x0<0) if !(y0<0) y2 = y0;
31 if !(x0<0) x2 = x0;
32 if !(x0<0) y3 = y2;
33 if (c0>0) c1 = c0 - 1;
34 if (c0>0) c2 = c1;
35 if !(c0>0) c2 = c0;
36 assert x2 >= 0;
37
38
39 // After third transformation step
40 if (x0<0) x1 = -x0;
41 if (x0<0) x2 = x1;
42 if (x0<0) y3 = y0;
43 if (!(x0<0 && y0<0) y1 = -y0;
44 if (!(x0<0 && y0<0) y2 = y1;
45 if (!(x0<0) && !(y0<0)) y2 = y0;
46 if !(x0<0) x2 = x0;
47 if !(x0<0) y3 = y2;
48 if (c0>0) c1 = c0 - 1;
49 if (c0>0) c2 = c1;
50 if !(c0>0) c2 = c0;
51 assert x2 >= 0;
```

Listing 3. A non-iterating program

The result of applying each of these steps to our running example program is shown in Listing 3. Note that the rules allow programs to be rewritten into equivalent ones consisting uniquely of sequences of conditional statements of the form **if** (b) C, with C an atomic (assignment, assume, or assert) statement (statements that are not inside any conditional can always be guarded with the true condition). This is the so-called *Conditional Normal Form* of a single-assignment program, and its relevance is that it can be converted to logic in a straightforward way. It suffices to construct two sets of formulas as follows:

Conditional normal form VCs with SAVER

The tool calculates a verification condition by symbolic execution from a single-assignment program when invoked with the switch `--cnf`. It can be pretty-printed with the `--print` switch; without this switch the VCs will be sent to the solver for checking. The result of both invocations is shown below.

```
$ ./saver.py --cnf --print progs/prog-sac.imp
TRANSFORMING TO CONDITIONAL NORMAL FORM...
GENERATING VCs...
OPERATIONAL ENCODING C:
(x0<0 ==> x1==0-x0)
(x0<0 ==> x2==x1)
(x0<0 ==> y3==y0)
((!x0<0 && y0<0) ==> y1==0-y0)
((!x0<0 && y0<0) ==> y2==y1)
((!x0<0 && !y0<0) ==> y2==y0)
(!x0<0 ==> x2==x0)
(!x0<0 ==> y3==y2)
(c0>0 ==> c1==c0-1)
(c0>0 ==> c2==c1)
(!c0>0 ==> c2==c0)
ASSERTIONS P:
VC 1 :
x2>=0


$ ./saver.py --cnf progs/prog-sac.imp
GENERATING BOUNDED MODEL CHECKING VCs...
VC 1 :
solving...
proved
```

—$\mathcal{C}$ is the set of formulas $\{b \rightarrow x = e \mid$ **if** (b) x = e is a statement in the program$\}$

—$\mathcal{P}$ is the set of formulas $\{b \rightarrow t \mid$ **if** (b) **assert** t is a statement in the program$\}$

The idea is that the formulas in $\mathcal{C}$ describe the operational contents of the program, whereas the formulas in $\mathcal{P}$ correspond to the properties expressed as asserts. A verification condition can then readily be obtained as the formula

$$\bigwedge \mathcal{C} \rightarrow \bigwedge \mathcal{P}$$

Note then that *a single* global verification condition is generated, regardless of the number of assert statements and execution paths that are present in the program. This is quite different from symbolic execution, which generates one VC for every path leading to every assert stament. The conditional normal form technique is able to work with all the assert statements in a single VC because these statements are encoded in a localized way, guarded by the path conditions that enable them to be executed. Since the formulas in $\mathcal{P}$ contain path information, the encoding works for assert statements placed arbitrarily in the program.

From the point of view of debugging it may be easier to split the above VC into a set of VCs, one for each assert in the program. The SAVER tool does precisely

Validity vs. Satisfiability

Verification conditions are defined as formulas whose *validity* ensures that the program is correct. But in the context of bounded model checking the property-checking problem was formulated in terms of *satisfiability*.

The original formulation was this: once the sets $\mathcal{C}$ and $\mathcal{P}$ have been constructed, the set of formulas $\mathcal{C} \cup \{\neg \bigwedge \mathcal{P}\}$ is checked for satisfiability; if it is *unsatisfiable* then the program is correct (no assertion is violated). Otherwise, each model satisfying the set corresponds to an execution of the program that violates some assertion.

Now note that this can of course be presented as a validity problem, which results in the verification condition formulation used in the present text.

$$\begin{aligned} & \mathsf{UNSAT}\left(\mathcal{C} \cup \{\neg \bigwedge \mathcal{P}\}\right) \\ \iff & \mathsf{UNSAT}\left(\bigwedge \mathcal{C} \wedge \neg \bigwedge \mathcal{P}\right) \\ \iff & \models \neg \bigwedge \mathcal{C} \vee \bigwedge \mathcal{P} \\ \iff & \models \bigwedge \mathcal{C} \rightarrow \bigwedge \mathcal{P} \end{aligned}$$

this: if $\mathcal{P}$ is the set $\{p_1, \ldots, p_n\}$, we have instead the set of VCs:

$$\left\{\bigwedge \mathcal{C} \rightarrow p_1, \ldots, \bigwedge \mathcal{C} \rightarrow p_n\right\}$$

It is crucial that the transformations involved in producing the conditional normal form do not duplicate elements of the program for each execution path encountered. Recall that the verification conditions generated by symbolic execution contain, for instance, 3 copies of the equality formula $c_1 = c_0 - 1$, since there are 3 execution paths arriving at the corresponding statement, and 6 copies of $x_2 \geq 0$ since there are 6 execution paths going through the assert statement. No such duplication occurs in the conditional normal form encoding. *Conditions* are duplicated for each atomic statement guarded by them, but the number of such statements is at most linear in the length of the program.

The box in page 13 shows how the SAVER tool can be used to both print and prove correctness of a program based on this technique. The box also shows the single verification condition generated by the tool for the program of Listing 2.

Formal Definition. The following recursive function transforms a program into CNF form. The first parameter is a formula corresponding to the *path condition* of the present statement: the conjunction of conditions that has to be evaluated to true to enable the statement to be executed.

$$\begin{aligned} \mathsf{toCNF}(\pi, \mathbf{skip}) &= \mathbf{if}\ \pi\ \mathbf{then\ skip} \\ \mathsf{toCNF}(\pi, x := e) &= \mathbf{if}\ \pi\ \mathbf{then}\ x := e \\ \mathsf{toCNF}(\pi, \mathbf{assume}\,\theta) &= \mathbf{if}\ \pi\ \mathbf{then\ assume}\,\theta \\ \mathsf{toCNF}(\pi, C_1\,;\, C_2) &= \mathsf{toCNF}(\pi, C_1)\,;\, \mathsf{toCNF}(\pi, C_2) \\ \mathsf{toCNF}(\pi, \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= \mathsf{toCNF}(\pi \wedge b, C^t)\,;\, \mathsf{toCNF}(\pi \wedge \neg b, C^f) \\ \mathsf{toCNF}(\pi, \mathbf{assert}\,\theta) &= \mathbf{if}\ \pi\ \mathbf{then\ assert}\,\theta \end{aligned}$$

It should be easy to understand that $\mathsf{toCNF}(\top, C)$ produces the conditional normal form of the SA program C, with no neeed for further rewriting. From this it is straightforward to obtain the sets $\mathcal{C}$ and $\mathcal{P}$. The following recursive function does precisely this:

$$\begin{aligned}
\mathsf{CP}(\textbf{if } b \textbf{ then skip}) &= (\emptyset, \emptyset) \\
\mathsf{CP}(\textbf{if } b \textbf{ then } x := e) &= (\{b \to x = e\}, \emptyset) \\
\mathsf{CP}(\textbf{if } b \textbf{ then assume } \theta) &= (\{b \to \theta\}, \emptyset) \\
\mathsf{CP}(C_1 \,;\, C_2) &= (\mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{P}_1 \cup \mathcal{P}_2) \\
&\quad \text{where} \\
&\quad (\mathcal{C}_1, \mathcal{P}_1) = \mathsf{CP}(C_1) \\
&\quad (\mathcal{C}_2, \mathcal{P}_2) = \mathsf{CP}(C_2) \\
\mathsf{CP}(\textbf{if } b \textbf{ then assert } \theta) &= (\emptyset, \{b \to \theta\})
\end{aligned}$$

These two functions can be fused into a single one that produces verification conditions from the initial SA program without explicitly constructing the CNF intermediate form. Let us then define this function $\mathsf{VC}^{\mathsf{cnf}}$ such that $\mathsf{VC}^{\mathsf{cnf}}(\pi, C) = \mathsf{CP}(\mathsf{toCNF}(\pi, C))$.

Definition 3 Conditional Normal Form VCs. *Given an SA program C, its* conditional normal form verification condition *is the formula $\bigwedge \mathcal{C} \to \bigwedge \mathcal{P}$, where $(\mathcal{C}, \mathcal{P}) = \mathsf{VC}^{\mathsf{cnf}}(\top, C)$.*

$$\begin{aligned}
\mathsf{VC}^{\mathsf{cnf}}(\pi, \textbf{skip}) &= (\emptyset, \emptyset) \\
\mathsf{VC}^{\mathsf{cnf}}(\pi, x := e) &= (\{\pi \to x = e\}, \emptyset) \\
\mathsf{VC}^{\mathsf{cnf}}(\pi, \textbf{assume } \theta) &= (\{\pi \to \theta\}, \emptyset) \\
\mathsf{VC}^{\mathsf{cnf}}(\pi, C_1 \,;\, C_2) &= (\mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{P}_1 \cup \mathcal{P}_2) \\
&\quad \textit{where} \\
&\quad (\mathcal{C}_1, \mathcal{P}_1) = \mathsf{VC}^{\mathsf{cnf}}(\pi, C_1) \\
&\quad (\mathcal{C}_2, \mathcal{P}_2) = \mathsf{VC}^{\mathsf{cnf}}(\pi, C_2) \\
\mathsf{VC}^{\mathsf{cnf}}(\pi, \textbf{if } b \textbf{ then } C^t \textbf{ else } C^f) &= (\mathcal{C}^t \cup \mathcal{C}^f, \mathcal{P}^t \cup \mathcal{P}^f) \\
&\quad \textit{where} \\
&\quad (\mathcal{C}^t, \mathcal{P}^t) = \mathsf{VC}^{\mathsf{cnf}}(\pi \wedge b, C^t) \\
&\quad (\mathcal{C}^f, \mathcal{P}^f) = \mathsf{VC}^{\mathsf{cnf}}(\pi \wedge \neg b, C^f) \\
\mathsf{VC}^{\mathsf{cnf}}(\pi, \textbf{assert } \theta) &= (\emptyset, \{\pi \to \theta\})
\end{aligned}$$

We will proceed to see in the next sections a third VC generation technique that will in a certain sense act as a bridge between the two techniques we have seen so far, but for now let us make here a brief remark on the nature of the accumulating parameters used in the recursive definitions corresponding to symbolic execution and conditional normal form. In the latter only *path conditions* are accumulated, i.e., formulas corresponding to the Boolean conditions that allow execution to reach the current statement, within the branching structure of the program. But in the former, the set Φ contains formulas encoding *execution paths*, i.e. formulas containing not only Boolean conditions but also information regarding variable assignments. This difference will be relevant later in this section.

Inclusion of asserts in the context

Consider the program

```
x1 := 0;
assert x1 > 10;
x2 := x1 + 10;
assert x2 > 20
```

It is clearly not a correct program, since both asserts are violated in any execution. But consider how the two verification techniques introduced thus far handle it:

```
$ ./saver.py --se progs/prog-context-assert.imp
GENERATING SYMBOLIC EXECUTION VCs...
VC 1 :
solving...
counterexample
[x1 = 0]
VC 2 :
solving...
proved

$ ./saver.py --cnf progs/prog-context-assert.imp
GENERATING CONDITIONAL NORMAL FORM VCs...
VC 1 :
solving...
counterexample
[x2 = 10, x1 = 0]
VC 2 :
solving...
counterexample
[x2 = 10, x1 = 0]
```

With CNF both VCs fail, whereas with symbolic execution the second VC is successfully discharged. To understand why, let us print the SE VCs:

```
./saver.py --se --print progs/prog-context-assert.imp
GENERATING SYMBOLIC EXECUTION VCs...
VC 1 :
(x1==0 ==> x1>10)
VC 2 :
(((x1==0 && x1>10) && x2==x1+10) ==> x2>20)
```

In the second VC the formula of the first assert command has been introduced in the context, resulting in a trivially valid VC, whereas with CNF the context never contains any assert formula. This should be clear from Definitions 2 and 3.

Note that the correctness of a program requires *all* VCs to be valid, and as such the SE technique is still sound. The inclusion of assert formulas in the context makes sense from the point of view of automated proof, since it allows assertions to be used as *lemmas*, intermediate formulas that are perhaps more easily proved than others, and when included in the context allow the latter to be proved. In the CNF technique the fact that the context (the set $\mathcal{C}$) is *the same for all VCs* precludes this possibility.

2.3 Predicate Transformers: Weakest Preconditions and Strongest Postconditions

There exist other approaches to the generation of verification conditions, based on the well-known notions of *weakest precondition* or (alternatively) of *strongest postcondition*. These approaches predate the techniques presented previously, and do not require converting programs to a single-assignment form. Instead, they are based on the propagation of assertions through the code, using variable substitutions to cope with assignment statements.

Weakest preconditions and strongest postconditions were introduced by Dijkstra as *predicate transformers* for his *guarded commands language* [Dijkstra 1976]. The language is different from the programming language we have been considering, in a number of ways. For instance it does not include a conditional construct, but includes instead a *non-deterministic choice* operator: the command $C_1 \,[\!]\, C_2$ will arbitrarily execute either C_1 or C_2. Conditionals can be encoded using the choice operator and assume commands for the guards. Although apparently standing at a higher level of abstraction than a standard imperative language, guarded commands are used in practice. In particular, they are the basis for the highly popular Boogie intermediate language and tool [Barnett et al. 2005].

In what follows we transpose the definition of weakest precondition and strongest postcondition to the language we have been considering. Given a command C and a formula ψ, the weakest precondition of C with respect to ψ, written $\mathsf{wp}(C, \psi)$, is the weakest formula such that, if the state in which C is executed satisfies this formula, then the execution is successful (no assert fails), and moreover ψ is true in the resulting state. Conversely, the strongest postcondition of C with respect to the formula ϕ, written $\mathsf{sp}(\phi, C)$, is the strongest formula that is satisfied after execution of C in any state satisfying ϕ.[2]

Consider the program

```
c = c-1;
assert c > 0;
```

Since we want to check whether c is positive after it has been decremented, we can calculate the weakest precondition $\mathsf{wp}(c := c - 1, c > 0)$. This is simply given as the result of substituting $c - 1$ for c in the formula $c > 0$ (in general, we have that $\mathsf{wp}(x := e, \psi)$ is given by $\psi[e/x]$). Naturally, the resulting formula $c-1 > 0 \equiv c > 1$ is satisfiable but not valid, which means that some executions will violate the assert statement. This formula is the verification condition for the program.

If the program was instead

```
c = 10;
c = c-1;
assert c > 0;
```

We would calculate $\mathsf{wp}(c := 10\,;\, c := c-1, c > 0)$ as being $\mathsf{wp}(c := 10, c-1 > 0)$, or simply $10 - 1 > 0$. Now this verification condition is valid, which means that every execution is safe. Note how the assertion $c > 0$ is propagated backwards through the program by weakest precondition computations: we calculate the weakest precondition of the first statement with respect to the weakest precondition of the second statement.

It is worth considering for a moment how this technique compares with the previous approaches. Whereas in those approaches the program was translated into a quasi-logical representation from which the VCs could be readily extracted, here the 'mismatch' between program variables and logic variables is handled by propagating assertions and performing substitution operations on them. The substitution operation eliminates the need to convert the program to a single-assignment form.

[2]In the presence of non-terminating constructs such as loops or recursion, these notions also force that execution always terminates.

VC by weakest precondition calculations:

$$
\begin{aligned}
&\mathsf{wp}(C_1\,;\,C_2, x \geq 0)\\
&= \mathsf{wp}(C_1, (c > 0 \to x \geq 0) \ \wedge\ (\neg c > 0 \to x \geq 0))\\
&= (x < 0 \to (c > 0 \to -x \geq 0) \ \wedge\ (\neg c > 0 \to -x \geq 0))\\
&\quad \wedge\ (\neg x < 0 \to \mathsf{wp}(\textbf{if } y < 0 \textbf{ then } y := -y \textbf{ else skip},\\
&\qquad\qquad (c > 0 \to x \geq 0) \ \wedge\ (\neg c > 0 \to x \geq 0)))\\
&= (x < 0 \to (c > 0 \to -x \geq 0) \ \wedge\ (\neg c > 0 \to -x \geq 0))\\
&\quad \wedge\ (\neg x < 0 \to (y < 0 \to (c > 0 \to x \geq 0) \ \wedge\ (\neg c > 0 \to x \geq 0))\\
&\qquad\qquad \wedge (\neg y < 0 \to (c > 0 \to x \geq 0) \ \wedge\ (\neg c > 0 \to x \geq 0)))
\end{aligned}
$$

VC rewritten by splitting:

$$
\begin{aligned}
&(x < 0 \to c > 0 \to -x \geq 0)\\
&\wedge\ (x < 0 \to \neg c > 0 \to -x \geq 0)\\
&\wedge\ (\neg x < 0 \to y < 0 \to c > 0 \to x \geq 0)\\
&\wedge\ (\neg x < 0 \to y < 0 \to \neg c > 0 \to x \geq 0)\\
&\wedge\ (\neg x < 0 \to \neg y < 0 \to c > 0 \to x \geq 0)\\
&\wedge\ (\neg x < 0 \to \neg y < 0 \to \neg c > 0 \to x \geq 0)
\end{aligned}
$$

Fig. 1. Weakest precondition of **ABS**

Weakest precondition computations are not more efficient than path enumeration by symbolic execution. To understand this, we have to consider how the weakest precondition of a conditional command is calculated. This will be given as a logical if-then-else, i.e. a formula $(b \to \omega_1) \wedge (\neg b \to \omega_2)$, where ω_1 and ω_2 are the weakest preconditions of both branches with respect to the desired assertion:

$$\mathsf{wp}(\textbf{if } b \textbf{ then } C^t \textbf{ else } C^f, \psi) = (b \to \mathsf{wp}(C^t, \psi)) \wedge (\neg b \to \mathsf{wp}(C^f, \psi))$$

Now consider that the program being checked is our proverbial sequence of n conditionals followed by the statement **assert** ψ. Propagating ψ through the last conditional will generate a formula containing two copies of ψ; this formula will then be propagated by each of the other conditionals, with all copies of ψ being duplicated at each step. Clearly, the resulting verification condition will contain 2^n copies of ψ; its size is thus exponential in the length of the program.

Figure 1 illustrates the computation of the weakest precondition VC for the example program of Listing 1. The reader will readily check that it contains 6 copies of the assertion $x \geq 0$, one for each execution path. To make this clearer, the figure also shows the result of rewriting the VC using the equivalence $\phi \to \psi_1 \wedge \psi_2 \ \equiv\ (\phi \to \psi_1) \wedge (\phi \to \psi_2)$, which results in a conjunctive formula where each conjunct corresponds to an execution path.

Strongest postcondition computations are less popular, even though the notion

itself is quite intuitive. Consider again the program

```
c = 10;
c = c-1;
assert c > 0;
```

The program is executed in an arbitrary state, described logically by the formula $\top$, so we start by propagating this formula forward. Clearly we will have $\mathsf{sp}(\top, \text{c=10}) \equiv c = 10$ and $\mathsf{sp}(\top, \text{c=10; c = c-1}) \equiv c = 9$. This should be fairly intuitive: the strongest postcondition summarizes logically what is known about the present state of execution of the program. Now that the assert statement has been reached, the verification condition can be written as the (valid) implicative formula $c = 9 \rightarrow c > 0$.

The reason why the strongest postcondition technique is less used than weakest precondition is simply that its definition introduces existential quantifiers, which are difficult to handle by automated theorem provers. In particular we have that $\mathsf{sp}(\phi, x := e)$ is given by $\exists\, x_0.\, \phi[x_0/x] \wedge x = e[x_0/x]$. Observe that unlike the notion of weakest precondition, the strongest postcondition must be able to refer to the values of variables before and after execution of the statement, and this is is done by introducing existentially quantified variables to represent the values in the pre-state.

Formal Definition. Let us write here functions to compute the weakest precondition and strongest postcondition of C with respect to a given assertion, adapted from the standard definitions of the corresponding predicate transformers in the guarded commands calculus.

Definition 4 Weakest Precondition. *Given a program C and an assertion ψ, the* weakest precondition $\mathsf{wp}(C, \psi)$ *of a program with respect to ψ can be calculated recursively as follows:*

$$\begin{aligned}
\mathsf{wp}(\mathbf{skip}, \psi) &= \psi \\
\mathsf{wp}(x := e, \psi) &= \psi[e/x] \\
\mathsf{wp}(\mathbf{assume}\,\theta, \psi) &= \theta \;\rightarrow\; \psi \\
\mathsf{wp}(C_1\,;\, C_2, \psi) &= \mathsf{wp}(C_1, \mathsf{wp}(C_2, \psi)) \\
\mathsf{wp}(\mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f, \psi) &= (b \rightarrow \mathsf{wp}(C^t, \psi)) \;\wedge\; (\neg b \rightarrow \mathsf{wp}(C^f, \psi)) \\
\mathsf{wp}(\mathbf{assert}\,\theta, \psi) &= \theta \;\wedge\; \psi
\end{aligned}$$

Definition 5 Strongest Postcondition. *Given a program C and an assertion ϕ, the* strongest postcondition *of C with respect to ϕ can be calculated as follows:*

$$\begin{aligned}
\mathsf{sp}(\phi, \mathbf{skip}) &= \phi \\
\mathsf{sp}(\phi, x := e) &= \exists\, x_0.\, \phi[x_0/x] \wedge x = e[x_0/x] \\
\mathsf{sp}(\phi, \mathbf{assume}\,\theta) &= \phi \;\wedge\; \theta \\
\mathsf{sp}(\phi, C_1\,;\, C_2) &= \mathsf{sp}(\mathsf{sp}(\phi, C_1), C_2) \\
\mathsf{sp}(\phi, \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= \mathsf{sp}(\phi \wedge b, C^t) \;\vee\; \mathsf{sp}(\phi \wedge \neg b, C^f) \\
\mathsf{sp}(\phi, \mathbf{assert}\,\theta) &= \phi \;\wedge\; \theta
\end{aligned}$$

We remark that while the weakest precondition is itself already a verification condition for a program (the only one required), the strongest postcondition is not in itself a VC. The function defined below constructs a set of verification conditions using strongest postcondition calculations. It takes an assertion as an accumulating parameter which, as was the case in symbolic execution, encodes the execution paths arriving at the present statement.

Definition 6 Strongest Postcondition VCs. *Given a program C, its* strongest postcondition verification conditions *are given by $\mathsf{VC}^{\mathsf{sp}}(\top, C)$, where the function $\mathsf{VC}^{\mathsf{sp}}$ is defined as follows.*

$$\begin{aligned}
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{skip}) &= \emptyset \\
\mathsf{VC}^{\mathsf{sp}}(\phi, x := e) &= \emptyset \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{assume}\,\theta) &= \emptyset \\
\mathsf{VC}^{\mathsf{sp}}(\phi, C_1\,;\,C_2) &= \mathsf{VC}^{\mathsf{sp}}(\phi, C_1) \;\cup\; \mathsf{VC}^{\mathsf{sp}}(\mathsf{sp}(\phi, C_1), C_2) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= \mathsf{VC}^{\mathsf{sp}}(\phi \wedge b, C^t) \;\cup\; \mathsf{VC}^{\mathsf{sp}}(\phi \wedge \neg b, C^f) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{assert}\,\theta) &= \{\phi \rightarrow \theta\}
\end{aligned}$$

Whereas in the symbolic execution algorithm the accumulator was a set of formulas, it is here a single formula (potentially of exponential size) computed by the strongest postcondition predicate transformer.

Let us step back to consider what we have seen up to this point. We started by looking at a VC generation technique that requires conversion of the programs to SA form followed by an enumeration of paths, potentially resulting in exponentially-sized verification conditions. We then saw how the conditional normal form technique (also based on conversion to SA form), avoided path enumeration and the risk of exponential explosion, by performing some simple transformations on the SA programs.

Finally, we looked at two techniques based on predicate transformers that compute VCs by propagating information backward (weakest precondition computations) or forward (strongest postcondition computations). The techniques avoid conversion to SA form by resorting to variable substitutions when propagating information, which in the case of strongest postcondition computations requires the introduction of existential quantifiers. These techniques are however amenable to exponential explosion, similarly to symbolic execution.

2.4 Efficient Predicate Transformers

The predicate transformer techniques can of course also be applied to programs in single-assignment form, which are just special cases of imperative programs. This may at first seem like a bad idea, since we have introduced conversion to SA form and the use of predicate transformers as *alternative* techniques to solve the same problem. However, it turns out that predicate transformers can, in the case of SA programs, be calculated in a simplified way that has an important consequence from the point of view of the practicality of program verification:

> *The exponential explosion in the size of the verification conditions is tamed – it becomes worst-case quadratic in the length of the program.*

The key point to understanding the simplified definition of predicate transformers for SA programs is to notice that the set of execution paths of such a program can be encoded logically in a compact way that does not require duplicating information. We call this encoding the *program formula*. The program formula of an assignment statement is simply the corresponding equality, and the formula of a sequence of statements is the conjunction of formulas of the substatements. For conditionals, this will be a formula $(b \wedge \omega_1) \vee (\neg b \wedge \omega_2)$, where ω_1 and ω_2 are the formulas of the branch statements. For instance the program

```
if (c0>0) {
  c1 = c0 - 1;
  c2 = c1;
}
else c2 = c0;
```

will be encoded by the formula $(c0 > 0 \wedge c1 = c0 - 1 \wedge c2 = c1) \vee (\neg c0 > 0 \wedge c2 = c0)$.

It turns out that the strongest postcondition with respect to a given assertion can be calculated by just combining the program formula with the assertion, *without further traversals of the program*. In particular we have

$$\mathsf{sp}(\phi, C) \equiv \phi \wedge \mathcal{F}(C)$$

so crucially, the assertions are not propagated through the program and thus not copied for each branch in it. Take for instance the simple program

```
c0 = 10;
c1 = c0-1;
assert c1 > 0;
```

We have $\mathsf{sp}(\top, \texttt{c0 = 10}) \equiv c0 = 10$ and $\mathsf{sp}(\top, \texttt{c0 = 10; c1 = c0-1}) \equiv c0 = 10 \wedge c1 = c0 - 1$. The VC calculated using strongest postcondition would be

$$\mathsf{sp}(\top, \texttt{c0 = 10; c1 = c0-1}) \rightarrow c1 > 0$$

or

$$c0 = 10 \wedge c1 = c0 - 1 \rightarrow c1 > 0$$

The box in page 22 shows how the SAVER tool can be used to both print and prove correctness of a program based on efficient strongest postcondition computations. The box also shows the verification condition generated by the tool for the SA program of Listing 2, and for the following program `prog-quadratic.imp`:

```
if (x0>0) then y1 := 1 else  y1 := 0 end;
assert y1 == 0 or y1 ==1;
if (x0>0) then y2 := 1 else  y2 := 0 end;
assert y2 == y1;
if (x0>0) then y3 := 1 else  y3 := 0 end;
assert y3 == y1
```

This program is designed to illustrate the worst-case size of the generated verification conditions. It has three asserts, each of which is preceded by a set of execution paths of exponential size in the length of the corresponding prefix of the program.

Efficient strongest postcondition with SAVER

The tool calculates verification conditions by strongest postcondition calculations from a single-assignment program when invoked with the switch `--sp`. They can be pretty-printed with the `--print` switch; without this switch the VCs will be sent to the solver for checking. The result of both invocations is shown below (slightly reformatted for the sake of readability).

```
$ ./saver.py --sp  --print progs/prog-sac.imp
GENERATING EFFICIENT STRONGEST POSTCONDITION VCs...
VC 1 :
(
  (((x0<0 && ((x1==0-x0 && x2==x1) && y3==y0))
|| (!x0<0 && ((((y0<0 && (y1==0-y0 && y2==y1))
            || (!y0<0 && y2==y0)) && x2==x0) && y3==y2)))
&&
   ((c0>0 && (c1==c0-1 && c2==c1))
|| (!c0>0 && c2==c0))
)
==> x2>=0)


$ ./saver.py --sp progs/prog-sac.imp
GENERATING EFFICIENT STRONGEST POSTCONDITION VCs...
VC 1 :
solving...
proved

./saver.py --sp --print progs/prog-quadratic.imp
GENERATING EFFICIENT STRONGEST POSTCONDITION VCs...
VC 1 :
(((x0>0 && y1==1) || (!x0>0 && y1==0)) ==> (y1==0 || y1==1))
VC 2 :
(((((x0>0 && y1==1) || (!x0>0 && y1==0)) && (y1==0 || y1==1))
&& ((x0>0 && y2==1) || (!x0>0 && y2==0))) ==> y2==y1)
VC 3 :
(((((((x0>0 && y1==1) || (!x0>0 && y1==0)) && (y1==0 || y1==1))
&& ((x0>0 && y2==1) || (!x0>0 && y2==0))) && y2==y1)
&& ((x0>0 && y3==1) || (!x0>0 && y3==0))) ==> y3==y1)
```

The generated VCs all have linear size in that length, and since the number of asserts is bounded by the length of the program, the size of the VCs cannot be asymptotically higher than quadratic.

Formal Definition. The definition of program formula should be clear from the previous explanation. The recursive function $\mathsf{VC}^{\mathsf{sp}}$ recursively constructs the set of verification conditions of a program based on an efficient strongest postcondition computation using the program formula.

DEFINITION 7. *The* program formula $\mathcal{F}(C)$ *of a single-assignment command is*

defined as follows:

$$\begin{aligned}
\mathcal{F}(\mathbf{skip}) &= \top \\
\mathcal{F}(x := e) &= x = e \\
\mathcal{F}(\mathbf{assume}\,\theta) &= \theta \\
\mathcal{F}(C_1\,;\,C_2) &= \mathcal{F}(C_1) \wedge \mathcal{F}(C_2) \\
\mathcal{F}(\mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= (b \wedge \mathcal{F}(C^t)) \vee (\neg b \wedge \mathcal{F}(C^f)) \\
\mathcal{F}(\mathbf{assert}\,\theta) &= \theta
\end{aligned}$$

Note how the program formula is the same for the assume and assert commands: since the assert has the effect of placing the formula in the context, it becomes part of the logical encoding of the program. The difference between both commands is that the latter generates a verification condition.

The reader may check that indeed $\mathsf{sp}(\phi, C) \equiv \phi \wedge \mathcal{F}(C)$ follows from definitions 5 and 7 for any single-assignment command C. Now in order to calculate efficiently the verification conditions of a program it suffices to modify Definition 6 replacing

$$\mathsf{VC}^{\mathsf{sp}}(\phi, C_1\,;\,C_2) = \mathsf{VC}^{\mathsf{sp}}(\phi, C_1) \ \cup\ \mathsf{VC}^{\mathsf{sp}}(\mathsf{sp}(\phi, C_1), C_2)$$

by

$$\mathsf{VC}^{\mathsf{sp}}(\phi, C_1\,;\,C_2) = \mathsf{VC}^{\mathsf{sp}}(\phi, C_1) \ \cup\ \mathsf{VC}^{\mathsf{sp}}(\phi \wedge \mathcal{F}(C_1), C_2)$$

The following definition does precisely this, but also calculates the program formula on-the-fly, by tupling it with the set of verification conditions.

Definition 8 Strongest Postcondition VCs. *Given an SA program C, its* strongest postcondition verification conditions *are given by the set V where $(F, V) = \mathsf{VC}^{\mathsf{sp}}(\top, C)$, and $\mathsf{VC}^{\mathsf{sp}}$ is the function defined as follows:*

$$\begin{aligned}
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{skip}) &= (\top, \emptyset) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, x := e) &= (x = e, \emptyset) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{assume}\,\theta) &= (\theta, \emptyset) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, C_1\,;\,C_2) &= (F_1 \wedge F_2, V_1 \cup V_2) \\
&\quad \textit{where} \\
&\quad (F_1, V_1) = \mathsf{VC}^{\mathsf{sp}}(\phi, C_1) \\
&\quad (F_2, V_2) = \mathsf{VC}^{\mathsf{sp}}(\phi \wedge F_1, C_2) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= ((b \wedge F^t) \vee (\neg b \wedge F^f), V^t \cup V^f) \\
&\quad \textit{where} \\
&\quad (F^t, V^t) = \mathsf{VC}^{\mathsf{sp}}(\phi \wedge b, C^t) \\
&\quad (F^f, V^f) = \mathsf{VC}^{\mathsf{sp}}(\phi \wedge \neg b, C^f) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{assert}\,\theta) &= (\theta, \{\phi \to \theta\})
\end{aligned}$$

Tupling eliminates redundancy in the calculation of the program formula.[3]

[3]It may have occurred to the reader that the first component in the tuple calculated could be the strongest postcondition $\phi \wedge \mathcal{F}(C)$ instead of just the program formula $\mathcal{F}(C)$. A careful consid-

Efficient weakest preconditions

The efficient technique for computing weakest preconditions was first outlined in 2001 by Flanagan and Saxe, in the context of work on the Extended Static Checker for Java [Flanagan and Saxe 2001]. The authors propose to convert code into a *passive form*, which is a static single assignment form without assignment statements (thus the name passive). Assignment information is instead encoded through the use of *assume* commands. Rustan Leino [Leino 2005] later gave a simplified presentation, showing that the technique could be seen as a special case of a weakest precondition computation, taking advantage of a property that holds for single-assignment code.

This so-called "dream property" is the following equivalence:

$$\mathsf{wlp}(C, \psi) \equiv \mathcal{F}(C) \to \psi$$

where $\mathsf{wlp}(C, \psi)$ denotes the *weakest liberal precondition* of C with respect to ψ, that is, the weakest formula that, if true in the state in which C is executed, will result in ψ holding in the state after execution, *if the asserts in C do not fail*.

The notion of (conservative) weakest precondition strengthens this weaker notion by forcing that no assert fails. In particular, $\mathsf{wp}(C, \top)$ is the weakest precondition describing states in which executing C results in all asserts executing without failing, thus we can write $\mathsf{wp}(C, \psi) \equiv \mathsf{wp}(C, \top) \wedge \mathsf{wlp}(C, \psi)$, and if C is in SA form, using the dream property:

$$\mathsf{wp}(C, \psi) \equiv \mathsf{wp}(C, \top) \wedge (\mathcal{F}(C) \to \psi)$$

The reader can easily check directly (the definition of weakest liberal precondition is not required) that this property follows from definitions 4 and 7 when C is SA.

Now note that in the context of our programs, where no postcondition is given explicitly, $\mathsf{wp}(C, \top)$ is precisely the verification condition for the program C. Let us spell this out as a function definition based on Definition 4, applying the dream property in the case of sequence:

$$\begin{aligned}
\mathsf{wptrue}(\mathbf{skip}) &= \top \\
\mathsf{wptrue}(x := e) &= \top \\
\mathsf{wptrue}(\mathbf{assume}\,\theta) &= \top \\
\mathsf{wptrue}(C_1\,;\,C_2) &= \mathsf{wptrue}(C_1) \;\wedge\; (\mathcal{F}(C_1) \to \mathsf{wptrue}(C_2)) \\
\mathsf{wptrue}(\mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= (b \to \mathsf{wptrue}(C^t)) \wedge\; (\neg b \to \mathsf{wptrue}(C^f)) \\
\mathsf{wptrue}(\mathbf{assert}\,\theta) &= \theta
\end{aligned}$$

The dream property allows for the VCs to be calculated avoiding the exponential pattern, since no duplication happens in the case of conditionals.

The first remark to be made here is that there is a striking similarity between the above definition and Definition 2, i.e. between the VC generation algorithms based on symbolic execution and on strongest postcondition computations. Both functions $\mathsf{VC}^{\mathsf{se}}$ and $\mathsf{VC}^{\mathsf{sp}}$ take an accumulating parameter corresponding to the logical encoding of the execution paths arriving at the present statement; the difference is that in $\mathsf{VC}^{\mathsf{sp}}$ these paths are encoded through a single compact formula $\mathcal{F}(C)$, without duplication when branching occurs, whereas for $\mathsf{VC}^{\mathsf{se}}$ the function paths

eration of the conditional clause reveals however that this would again fall into the exponential explosion trap, since there would be no way to factor out the propagated (and duplicated) formula ϕ.

constructs individual formulas explicitly for each execution path. This relationship can be expressed formally as follows:

PROPOSITION 9. *Let C be an SA program, Φ a set of formulas, and $(F, V) = \mathsf{VC}^{\mathsf{sp}}(\bigwedge \Phi, C)$. Then*

(1) $\quad \bigwedge \mathsf{paths}(\Phi, C) \;\equiv\; F$

(2) $\quad \models \mathsf{VC}^{\mathsf{se}}(\Phi, C) \;\; \textit{iff} \;\; \models V$

PROOF. Both equivalences are proved by induction on the structure of the program C. □

This result establishes an 'extensional' equivalence between the sets of VCs generated by symbolic execution and by strongest postcondition computations, but clearly the latter method is preferable since the size of the VCs generated by symbolic execution is potentially exponential. The significance of this result is that, although we have not formally proved the soundness of any of the techniques, we now know that the soundness of one of the techniques implies the soundness of the other.

To finish this section, let us consider how the verification conditions generated using efficient strongest postcondition calculations relate to those using efficient weakest preconditions. In order to do this we modify the definition of the function wptrue in the box of page 24 to produce a set of VCs (one for each assert) instead of a single formula, as follows:

$$\begin{aligned}
\mathsf{VC}^{\mathsf{wp}}(\mathbf{skip}) &= \emptyset \\
\mathsf{VC}^{\mathsf{wp}}(x := e) &= \emptyset \\
\mathsf{VC}^{\mathsf{wp}}(\mathbf{assume}\,\theta) &= \emptyset \\
\mathsf{VC}^{\mathsf{wp}}(C_1\,;\,C_2) &= \mathsf{VC}^{\mathsf{wp}}(C_1) \;\cup\; (\mathcal{F}(C_1) \rightarrow \mathsf{VC}^{\mathsf{wp}}(C_2)) \\
\mathsf{VC}^{\mathsf{wp}}(\mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= (b \rightarrow \mathsf{VC}^{\mathsf{wp}}(C^t)) \cup\; (\neg b \rightarrow \mathsf{VC}^{\mathsf{wp}}(C^f)) \\
\mathsf{VC}^{\mathsf{wp}}(\mathbf{assert}\,\theta) &= \{\theta\}
\end{aligned}$$

where we use the notation $\phi \rightarrow \Gamma$ to denote the set $\{\phi \rightarrow \gamma \mid \gamma \in \Gamma\}$. Then we have the following:

LEMMA 10. *Let C be an SA program, ϕ an assertion, and $(F, V) = \mathsf{VC}^{\mathsf{sp}}(\phi, C)$; then*

$$V = \phi \rightarrow \mathsf{VC}^{\mathsf{wp}}(C)$$

PROOF. By induction on the structure of C, using definition 8 and the above definition of $\mathsf{VC}^{\mathsf{wp}}$. □

Interestingly, this definition of verification conditions based on efficient weakest precondition calculations does not require a second parameter to accumulate execution paths or path conditions (observe that it could also calculate the program formula on-the-fly and return a tuple, similarly to the definition of $\mathsf{VC}^{\mathsf{sp}}$).

$$
\begin{array}{l}
\quad (x0 < 0 \wedge (x1 = -x0 \wedge x2 = x1 \wedge y3 = y0)) \\
\vee\ (\neg x0 < 0 \wedge (((y0 < 0 \wedge (y1 = -y0 \wedge y2 = y1)) \vee (\neg y0 < 0 \wedge y2 = y0)) \wedge x2 = x0 \wedge y3 = y2)) \\
\\
\equiv\ (1) \\
\\
\quad (x0 < 0 \rightarrow (x1 = -x0 \wedge x2 = x1 \wedge y3 = y0)) \\
\wedge\ (\neg x0 < 0 \rightarrow (((y0 < 0 \rightarrow (y1 = -y0 \wedge y2 = y1)) \wedge (\neg y0 < 0 \rightarrow y2 = y0)) \wedge x2 = x0 \wedge y3 = y2)) \\
\\
\equiv\ (2) \\
\\
\quad ((x0 < 0 \rightarrow x1 = -x0) \wedge (x0 < 0 \rightarrow x2 = x1) \wedge (x0 < 0 \rightarrow y3 = y0)) \\
\wedge\ (\neg x0 < 0 \rightarrow ((y0 < 0 \rightarrow y1 = -y0) \wedge (y0 < 0 \rightarrow y2 = y1) \wedge (\neg y0 < 0 \rightarrow y2 = y0) \wedge x2 = x0 \wedge y3 = y2)) \\
\\
\equiv\ (2) \\
\\
\quad (x0 < 0 \rightarrow x1 = -x0) \\
\wedge\ (x0 < 0 \rightarrow x2 = x1) \\
\wedge\ (x0 < 0 \rightarrow y3 = y0) \\
\wedge\ (\neg x0 < 0 \rightarrow y0 < 0 \rightarrow y1 = -y0) \\
\wedge\ (\neg x0 < 0 \rightarrow y0 < 0 \rightarrow y2 = y1) \\
\wedge\ (\neg x0 < 0 \rightarrow \neg y0 < 0 \rightarrow y2 = y0) \\
\wedge\ (\neg x0 < 0 \rightarrow x2 = x0) \\
\wedge\ (\neg x0 < 0 \rightarrow y3 = y2) \\
\\
\equiv\ (3) \\
\\
\quad (x0 < 0 \rightarrow x1 = -x0) \\
\wedge\ (x0 < 0 \rightarrow x2 = x1) \\
\wedge\ (x0 < 0 \rightarrow y3 = y0) \\
\wedge\ (\neg x0 < 0 \wedge y0 < 0 \rightarrow y1 = -y0) \\
\wedge\ (\neg x0 < 0 \wedge y0 < 0 \rightarrow y2 = y1) \\
\wedge\ (\neg x0 < 0 \wedge \neg y0 < 0 \rightarrow y2 = y0) \\
\wedge\ (\neg x0 < 0 \rightarrow x2 = x0) \\
\wedge\ (\neg x0 < 0 \rightarrow y3 = y2)
\end{array}
$$

Fig. 2. Rewriting strongest postcondition VCs

2.5 Splitting Strongest Postcondition

Let us now consider a variant of the previous algorithm for obtaining verification conditions using postcondition computations. We start precisely from the VCs obtained by the function $\mathsf{VC}^{\mathsf{sp}}$, and then rewrite them using the following simple equivalences. Equivalence 1 concerns two standard encodings of "if-then-else"; equivalences 2 and 3 are also both well-known.

$$(\pi \wedge \alpha) \vee (\neg\pi \wedge \beta) \equiv (\pi \rightarrow \alpha) \wedge (\neg\pi \rightarrow \beta) \tag{1}$$

$$\alpha \rightarrow \beta \wedge \gamma \equiv (\alpha \rightarrow \beta) \wedge (\alpha \rightarrow \gamma) \tag{2}$$

$$\alpha \rightarrow \beta \rightarrow \gamma \equiv \alpha \wedge \beta \rightarrow \gamma \tag{3}$$

To understand the effect of this rewriting on the VCs, we take in Figure 2 a formula corresponding to the efficient strongest postcondition encoding of the first part of our example program, and apply the above equivalences to it. It can readily be understood that after rewriting conditionals as conjunctions by equivalence 1, equivalence 2 allows the path conditions to be distributed relative to the atomic commands, and equivalence 3 groups these conditions into conjunctive formulas, resulting in a similar encoding to the one obtained by rewriting the program to

Splitting strongest postcondition with SAVER

The tool calculates verification conditions by splitting strongest postcondition calculations from a single-assignment program when invoked with the switch `--splitsp`. They can be pretty-printed with the `--print` switch; without this switch the VCs will be sent to the solver for checking. The result of both invocations is shown below (reformatted for the sake of readability).

```
$ ./saver.py --splitsp --print progs/prog-sac.imp
GENERATING SPLIT EFFICIENT STRONGEST POSTCONDITION VCs...
VC 1 :
((((((((((((x0<0 ==> x1==0-x0)
        && (x0<0 ==> x2==x1))
        && (x0<0 ==> y3==y0))
        && ((!x0<0 && y0<0) ==> y1==0-y0))
        && ((!x0<0 && y0<0) ==> y2==y1))
        && ((!x0<0 && !y0<0) ==> y2==y0))
        && (!x0<0 ==> x2==x0))
        && (!x0<0 ==> y3==y2))
        && (c0>0 ==> c1==c0-1))
        && (c0>0 ==> c2==c1))
        && (!c0>0 ==> c2==c0))
==> x2>=0)


$ ./imp2.py --splitsp  progs/prog-sac.imp
GENERATING SPLIT EFFICIENT STRONGEST POSTCONDITION VCs...
VC 1 :
solving...
proved
```

conditional normal form (the only difference being that here a single conjunctive formula is obtained, rather than a set of formulas).

The box in page 27 shows how the SAVER tool can be used to both print and prove correctness of a program based on strongest postcondition computations rewritten in the way described. We will call this the *splitting strongest postcondition* verification conditions. The box also shows the verification condition generated by the tool for the SA program of Listing 2.

Formal Definition. The same effect of applying rewriting with the equivalences given above can be obtained by simply modifying the function that computes the program formula.

DEFINITION 11. *The* splitting program formula *of a single-assignment command*

C is given by $\mathcal{F}^s(\top, C)$, where the function $\mathcal{F}^s$ is defined as follows:

$$\begin{aligned}
\mathcal{F}^s(\pi, \mathbf{skip}) &= \top \\
\mathcal{F}^s(\pi, x := e) &= \pi \to x = e \\
\mathcal{F}^s(\pi, \mathbf{assume}\,\theta) &= \pi \to \theta \\
\mathcal{F}^s(\pi, C_1\,;\, C_2) &= \mathcal{F}^s(\pi, C_1) \ \wedge\ \mathcal{F}^s(\pi, C_2) \\
\mathcal{F}^s(\pi, \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= \mathcal{F}^s(\pi \wedge b, C^t) \wedge \mathcal{F}^s(\pi \wedge \neg b, C^f) \\
\mathcal{F}^s(\pi, \mathbf{assert}\,\theta) &= \pi \to \theta
\end{aligned}$$

Observe that the accumulator parameter π corresponds here to a *path condition* formula and not to an execution path formula, since in the $\mathcal{F}^s(\pi, C_1\,;\, C_2)$ clause the recursive call $\mathcal{F}^s(\pi, C_2)$ receives the unmodified π as parameter. The function accumulates path conditions recursively, and uses these conditions to construct implicative formulas when an atomic statement is met.

PROPOSITION 12. *Let C be an SA program and π a formula; then*

$$\mathcal{F}^s(\pi, C) \ \equiv\ \pi \to \mathcal{F}(C)$$

PROOF. By induction on the structure of the program C, using definitions 7 and 11, and equivalences (1) to (3). □

As a consequence of this lemma we could obtain a VC generator equivalent to the one of Definition 8 by simply replacing the second recursive call in the sequence case by $\mathsf{VC}^{\mathsf{sp}}(\phi \wedge \mathcal{F}^s(\top, C_1), C_2)$. But let us write instead a tupling definition that computes the program formula and the VCs simultaneously.

To do this, first note that the function $\mathcal{F}^s$ carries a path formula as parameter, whereas $\mathsf{VC}^{\mathsf{sp}}$ requires a full program formula. We will thus separate the path formula component of the program formula, and write a function with two parameters: a path formula π, and a formula γ such that $\gamma \wedge \pi$ is the program formula taken as argument by $\mathsf{VC}^{\mathsf{sp}}$.

DEFINITION 13 SPLIT STRONGEST POSTCONDITION VCS. *Given an SA program C, its* split strongest postcondition verification conditions *are given by the set V where $(F, V) = \mathsf{VC}^{\mathsf{ssp}}(\top, \top, C)$, and $\mathsf{VC}^{\mathsf{ssp}}$ is defined as follows:*

$$\begin{aligned}
\mathsf{VC}^{\mathsf{ssp}}(\pi, \gamma, \mathbf{skip}) &= (\top, \emptyset) \\
\mathsf{VC}^{\mathsf{ssp}}(\pi, \gamma, x := e) &= (\pi \to x = e, \emptyset) \\
\mathsf{VC}^{\mathsf{ssp}}(\pi, \gamma, \mathbf{assume}\,\theta) &= (\pi \to \theta, \emptyset) \\
\mathsf{VC}^{\mathsf{ssp}}(\pi, \gamma, C_1\,;\, C_2) &= (F_1 \wedge F_2, V_1 \cup V_2) \\
&\quad \textit{where} \\
&\quad (F_1, V_1) = \mathsf{VC}^{\mathsf{ssp}}(\pi, \gamma, C_1) \\
&\quad (F_2, V_2) = \mathsf{VC}^{\mathsf{ssp}}(\pi, \gamma \wedge F_1, C_2) \\
\mathsf{VC}^{\mathsf{ssp}}(\pi, \gamma, \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= (F^t \wedge F^f, V^t \cup V^f) \\
&\quad \textit{where} \\
&\quad (F^t, V^t) = \mathsf{VC}^{\mathsf{ssp}}(\pi \wedge b, \gamma, C^t) \\
&\quad (F^f, V^f) = \mathsf{VC}^{\mathsf{ssp}}(\pi \wedge \neg b, \gamma, C^f) \\
\mathsf{VC}^{\mathsf{ssp}}(\pi, \gamma, \mathbf{assert}\,\theta) &= (\pi \to \theta, \{\gamma \to \pi \to \theta\})
\end{aligned}$$

This definition results in a set of VCs equivalent to those given by Definition 8:

PROPOSITION 14. *Let C be an SA program, π, γ, and ϕ assertions such that $\phi \equiv \gamma \wedge \pi$, $(F, V) = \mathsf{VC}^{\mathsf{sp}}(\phi, C)$ and $(F^s, V^s) = \mathsf{VC}^{\mathsf{ssp}}(\pi, \gamma, C)$; then*

(1) $F^s \equiv \pi \rightarrow F$

(2) $V \models V^s$ *and* $V^s \models V$, *i.e.* V *and* V^s *are equivalent sets of VCs.*

PROOF. *By induction on the structure of* C. □

2.6 A Generalized Efficient VCGen

What is interesting about the above split version of strongest postcondition VCs is that they are very close to those produced by the CNF transformation. We will now write a generalized VCGen that subsumes both VCGens of definitions 3 and 13. This generalized VCGen can be fine-tuned by just changing one clause, as explained below.

DEFINITION 15. *Given an SA program C, its* generalized efficient verification conditions *are given by the set $\mathcal{P}$ where $(\mathcal{C}, \mathcal{P}) = \mathsf{VC}^{\mathsf{gen}}(\top, \emptyset, C)$, and $\mathsf{VC}^{\mathsf{gen}}$ is defined as follows. Note that three different versions are given, depending on which clause is chosen in the* assert *case.*

$$
\begin{aligned}
\mathsf{VC}^{\mathsf{gen}}(\pi, \mathcal{C}, \mathbf{skip}) &= (\emptyset, \emptyset) \\
\mathsf{VC}^{\mathsf{gen}}(\pi, \mathcal{C}, x := e) &= (\{\pi \rightarrow x = e\}, \emptyset) \\
\mathsf{VC}^{\mathsf{gen}}(\pi, \mathcal{C}, \mathbf{assume}\,\theta) &= (\{\pi \rightarrow \theta\}, \emptyset) \\
\mathsf{VC}^{\mathsf{gen}}(\pi, \mathcal{C}, C_1\,;\, C_2) &= (\mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{P}_1 \cup \mathcal{P}_2) \\
&\quad \textit{where} \\
&\quad (\mathcal{C}_1, \mathcal{P}_1) = \mathsf{VC}^{\mathsf{gen}}(\pi, \mathcal{C}, C_1) \\
&\quad (\mathcal{C}_2, \mathcal{P}_2) = \mathsf{VC}^{\mathsf{gen}}(\pi, \mathcal{C} \cup \mathcal{C}_1, C_2) \\
\mathsf{VC}^{\mathsf{gen}}(\pi, \mathcal{C}, \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= (\mathcal{C}^t \cup \mathcal{C}^f, \mathcal{P}^t \cup \mathcal{P}^f) \\
&\quad \textit{where} \\
&\quad (\mathcal{C}^t, \mathcal{P}^t) = \mathsf{VC}^{\mathsf{gen}}(\pi \wedge b, \mathcal{C}, C^t) \\
&\quad (\mathcal{C}^f, \mathcal{P}^f) = \mathsf{VC}^{\mathsf{gen}}(\pi \wedge \neg b, \mathcal{C}, C^f) \\
\mathsf{VC}^{\mathsf{gen}}(\pi, \mathcal{C}, \mathbf{assert}\,\theta) &= \begin{cases} (\{\pi \rightarrow \theta\}, \{\bigwedge \mathcal{C} \rightarrow \pi \rightarrow \theta\}) & (1) \\ (\{\}, \{\bigwedge \mathcal{C} \rightarrow \pi \rightarrow \theta\}) & (2) \\ (\{\}, \{\pi \rightarrow \theta\}) & (3) \end{cases}
\end{aligned}
$$

First of all, we note that with clause (1) the above definition corresponds exactly to the split postcondition VCGen of Definition 13, except that the computational formulas γ and F are replaced by contexts (sets of formulas $\mathcal{C}$). Now recall that in the box of page 16 we have identified a characteristic of the CNF encoding that can be seen as a limitation: the context $\mathcal{C}$ is the same one for every assert formula in the program. The above definition with clause (1) can also be seen as a modification of the CNF VCGen that incorporates specific partial contexts for each assert, encoding only the relevant part of the program.

Partial contexts have (in principle) two advantages: the formulas sent to the solver are streamlined (the irrelevant information that would be included in a global

```
1  if (x0 < 0) x1 = -x0;
2  else {
3    if (y0 < 0) y1 = -y0;
4    y2 = (y0 < 0) ? y1 : y0;
5  }
6  x2 = (x0 < 0) ? x1 : x0;
7  y3 = (x0 < 0) ? y0 : y2;
8  if (c0 > 0) c1 = c0 - 1;
9  c2 = (c0 > 0) ? c1: c0;
10 assert x2 >= 0;
11
12 // After transformation
13 x1 = -x0;
14 y1 = -y0;
15 y2 = (y0 < 0) ? y1 : y0;
16 x2 = (x0 < 0) ? x1 : x0;
17 y3 = (x0 < 0) ? y0 : y2;
18 c1 = c0 - 1;
19 c2 = (c0 > 0) ? c1: c0;
20 assert x2 >= 0;
```

Listing 4. Example program in SSA form with conditional expressions

context is omitted); and assert formulas can be included as lemmas in the contexts for proving other asserts that occur subsequently in the code. Clause (1) does precisely this. Clause (2) on the other hand introduces a variant: partial contexts are present, but do not include previous assert formulas (this is the semantics implemented in the CBMC bounded model checker [Clarke et al. 2004]). Finally, clause (3) of the definition gives us the single-context CNF VCGen of Definition 3: no partial contexts are calculated; the VCs have to be proved in the global context (calculated as the first component of the pair).

2.7 VC Generation from SSA Code

Recall from the box of page 9 that the single-assignment forms we have been considering are dynamic: the single-assignment restrictions apply to executions of the program, so the same variable is allowed to be assigned in different execution paths. In bounded model checking of software static single-assignment forms have been more popular, making use of conditional expressions to encode Φ functions.

Listing 4 (top) shows what the example of Listing 1 would look like in SSA form. We remark that formulas containing conditional ("if-then-else") expressions are straightforward to encode logically, and can in fact be written directly in the SMT-LIB language. With this in mind, any of the VCGen algorithms studied in the previous sections can be applied to code in SSA form, with no modifications.

There exists however an additional point of interest in the use of static SA forms, which is the fact that a simpler logical encoding can be obtained by omitting a significant part of the control flow information. In particular, assignment commands can be encoded without their path conditions, since all the required information is contained in the conditional expressions that are assigned to the variables used to synchronize conditional branches. A program can thus be transformed into a much

simpler SSA program that, although not operationally equivalent to the original one, keeps the necessary information to generate simplified verification conditions.

The result of the simplification for our example program is shown in Listing 4 (bottom), and it can readily be appreciated how the result compares with the code generated by the CNF transformation (see Listing 3): the assignment statements are basically the same, but some conditional expressions have been eliminated, and others have had their conditions simplified (conjuncts were dropped).

Of course, similarly to the CNF transformation of Section 3, it is possible to write an algorithm for VC generation that does not explicitly transform the program. The following definition is similar to the generalized VCGen of Definition 15 (in particular in what concerns the different options for partial or global contexts), but incorporates the simplification described above.

DEFINITION 16 VCs FOR SSA PROGRAMS. *Given an SSA program C, its VCs are given by the set $\mathcal{P}$, where $(\mathcal{C}, \mathcal{P}) = \mathsf{VC}^{\mathsf{ssa}}(\top, \emptyset, C)$, and $\mathsf{VC}^{\mathsf{ssa}}$ is defined as follows. Again three different versions are given, depending on which clause is chosen in the* assert *case.*

$$
\begin{aligned}
\mathsf{VC}^{\mathsf{ssa}}(\pi, \mathcal{C}, \mathbf{skip}) &= (\emptyset, \emptyset) \\
\mathsf{VC}^{\mathsf{ssa}}(\pi, \mathcal{C}, x := e) &= (\{x = e\}, \emptyset) \\
\mathsf{VC}^{\mathsf{ssa}}(\pi, \mathcal{C}, \mathbf{assume}\,\theta) &= (\{\pi \to \theta\}, \emptyset) \\
\mathsf{VC}^{\mathsf{ssa}}(\pi, \mathcal{C}, C_1\,;\, C_2) &= (\mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{P}_1 \cup \mathcal{P}_2) \\
&\quad \textit{where} \\
&\quad (\mathcal{C}_1, \mathcal{P}_1) = \mathsf{VC}^{\mathsf{ssa}}(\pi, \mathcal{C}, C_1) \\
&\quad (\mathcal{C}_2, \mathcal{P}_2) = \mathsf{VC}^{\mathsf{ssa}}(\pi, \mathcal{C} \cup \mathcal{C}_1, C_2) \\
\mathsf{VC}^{\mathsf{ssa}}(\pi, \mathcal{C}, \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= (\mathcal{C}^t \cup \mathcal{C}^f, \mathcal{P}^t \cup \mathcal{P}^f) \\
&\quad \textit{where} \\
&\quad (\mathcal{C}^t, \mathcal{P}^t) = \mathsf{VC}^{\mathsf{ssa}}(\pi \wedge b, \mathcal{C}, C^t) \\
&\quad (\mathcal{C}^f, \mathcal{P}^f) = \mathsf{VC}^{\mathsf{ssa}}(\pi \wedge \neg b, \mathcal{C}, C^f) \\
\mathsf{VC}^{\mathsf{ssa}}(\pi, \mathcal{C}, \mathbf{assert}\,\theta) &= \begin{cases} (\{\pi \to \theta\}, \{\bigwedge \mathcal{C} \to \pi \to \theta\}) & (1) \\ (\{\}, \{\bigwedge \mathcal{C} \to \pi \to \theta\}) & (2) \\ (\{\}, \{\pi \to \theta\}) & (3) \end{cases}
\end{aligned}
$$

We end with the following note: for the purpose of mapping errors back to the original source code it may be useful to assign conditions explicitly to Boolean variables, as shown in Listing 5. This does not affect the VC generation algorithm.

2.8 Empirical Comparison of VC Generation Techniques

As far as we know the different VC generation methods presented previously have not been compared empirically. Although for small-sized formulas the difference may well be irrelevant, verification techniques like bounded model checking generate huge formulas (see Section 5.1), and it is perfectly possible that different logical encodings will produce very different results concerning the performance of the satisfiability solvers.

```
1  b1 = x0<0;
2  if (b1) x1 = -x0;
3  else {
4    b2 = y0<0;
5    if (b2) y1 = -y0;
6    y2 = b2 ? y1 : y0;
7  }
8  x2 = b1 ? x1 : x0;
9  y3 = b1 ? y0 : y2;
10 b3 = c0>0;
11 if (b3) c1 = c0 - 1;
12 c2 = b3 ? c1: c0;
13 assert x2 >= 0;
14
15 // After transformation
16 b1 = x0<0;
17 x1 = -x0;
18 b2 = y0<0;
19 y1 = -y0;
20 y2 = b2 ? y1 : y0;
21 x2 = b1 ? x1 : x0;
22 y3 = b1 ? y0 : y2;
23 b3 = c0>0;
24 c1 = c0 - 1;
25 c2 = b3 ? c1: c0;
26 assert x2 >= 0;
```

Listing 5. Example program in SSA form with conditional expressions and Boolean variables, and the same code in flattened form

In this section we have basically arrived at two VCGen algorithms (Definitions 8 and 15) that produce different encodings of the branching structure of programs. One of these admits variants regarding contexts and the insertion of assert formulas as lemmas; these algorithms would have to be tested with different kinds of code, in particular code containing intermediate assert commands.

But there is another dimension for comparison: code may be encoded based on a static or a dynamic notion of single-assignment, and the same VCGen algorithm should be applied to different SA encodings of the same code, to clarify if one particular encoding is always the best, or if some encoding works better with one VCGen algorithm but not with the other. Recall also that for SSA code one of the VCGens can be optimized as shown in Section 2.7; the effect of this should also be evaluated.

Finally, we remark that the SSA encoding is very close to the DSA encoding that was used in the examples in this section: comparing Listings 4 and 2, it is clear that both use the same set of variables, the difference being only that in the SSA encoding variables from different branches are joined outside the conditional, in a single assignment instruction. However, the DSA encoding can be optimized to use less variables, since each one may occur assigned more than once in different paths, and it is not necessary to introduce fresh version variables when joining conditional branches. Listing 6 shows the result of applying this 'efficient' DSA encoding to

```
1  if (x0<0) {
2    x1 := -x0;
3    y1 := y0
4  }
5  else {
6    if (y0<0) y1 := -y0
7    else y1 := y0;
8    x1 := x0
9  }
10 if (c0>0) c1 := c0 - 1
11 else c1 := c0;
12 assert x1 >= 0
```

Listing 6. Example program in efficient DSA form

our running example program. The effect of this improved use of variables on the performance of solvers should also be evaluated empirically.

3. HOARELOGIC AND VERIFICATION CONDITIONS

In the previous section we introduced various verification condition generation mechanisms for branching single-assignment programs. We appealed to the reader's intuitions about the semantics of imperative programs, without providing any proof of soundness or completeness for these mechanisms. Let us clarify what we mean by soundness: the sets of VCs generated by the mechanisms should be such that, if all the conditions are *valid*, then the program from which they were generated is *correct*, i.e. *no execution exists in which an assert statement fails*. Completeness corresponds to the reverse implication: the verification conditions generated from a correct program should be valid.

We will now change our focus to the setting of standard (non-single-assignment) programs, and review the fundamental framework used as a basis for program verification. A crucial part of the framework is *Hoare logic*, the seminal system for reasoning axiomatically about the behavior of imperative programs. We will show how a VC generation mechanism can be synthesized for standard programs, and shown to be sound and complete.

Hoare logic was proposed to deal with iterating While programs, based on the fundamental idea of loop invariant. Working verification tools expect users to provide loop invariants, usually as annotations in the code. There is a certain mismatch between Hoare logic and the functioning of these tools, since in the logic the invariants are invented as the derivations are constructed; they are not given beforehand. The framework reviewed here is intended as underlying the functioning of a program verification tool, and introduces the notions of annotated and correctly-annotated program, as well as a goal-directed program logic that makes it easier to express strategies for generating VCs. A more detailed study can be found in [Frade and Pinto 2011].

One novelty of this section is the treatment of assume and assert statements, which are not usually considered as part of a While language, but are typically included in the specification languages employed by tools such as SPARK, Frama-

C, tools for checking Java code based on the Java Modeling Language (JML), and of course software model checkers in general.

This framework will be adapted in the next section to prove soundness and completeness of one of the VCGens for single-assignment branching programs introduced in Section 2, and this is in fact one of the reasons why it includes assume and assert commands, which were present in the language of that section.

3.1 Operational Semantics of While Programs

We wish to verify programs in a formal setting, and the tools we use come from *programming language semantics*. In order to formalize the idea of a program execution failing, we first resort to an *operational semantics* of our programming language with assert and assume statements, with the following syntax:

$$\begin{array}{rcl} \mathbf{Comm} \ni C & ::= & \mathbf{skip} \mid C\,;\,C \mid x := e \mid \mathbf{assume}\,\theta \mid \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \\ & & \mathbf{while}\ b\ \mathbf{do}\ C \mid \mathbf{assert}\,\theta \end{array}$$

where b and e range respectively over Boolean expressions and program expressions. For now we consider that θ also ranges over Boolean expressions, but see the discussion in the box of page 38.

There are various styles of operational semantics; we will use what is usually called a *natural* or *evaluation semantics*, sometimes also called a *big-step* semantics. A natural semantics is given by defining the *evaluation relation* $\rightsquigarrow$ describing the final program states that may result from running a program in a given initial state. We will write $\langle C, s\rangle \rightsquigarrow s'$ to denote the fact that execution of C in state s terminates in state s'. A *program state* is simply a function that gives the value of the program variables. Given a set of variables **Var** and a domain of interpretation D for the program expressions, we have the set of program states

$$\Sigma = \mathbf{Var} \to D$$

State functions are defined for all variables (they are total functions).

The first ingredient of an evaluation semantics is *expression evaluation*. Given $s \in \Sigma$, $[\![e]\!](s)$ will denote the value of expression e in state s. In this section we will leave the language of expressions (and its types) unspecified, for the sake of generality. We simply assume that expression evaluation is deterministic and does not produce side effects (i.e. the state s remains unchanged after evaluation of e). Moreover, we will consider that expression evaluation does not produce errors; in particular the evaluation of all arithmetic expressions, including division by zero, produces some fixed value (the only notion of failure will result from the execution of assert statements).

It is not common to find an operational semantics of a programming language containing commands like assert and assume, which are used for specification, rather than operational, purposes.[4]

[4]These commands are present in Dijkstra's guarded commands language, where the semantics are given directly by the definition of a predicate transformer like *weakest precondition*, which describes their operational behavior implicitly rather than directly. What we seek to do here is to prove soundness and completeness of VC generation in a standard semantic setting, with a clear separation between different kinds of semantics. We will first introduce an operational semantics

We face a choice here between two approaches:

—To treat assert and assume as producing no observable effect on the execution of a program, and assign meaning to them later, when defining the notion of correct program; or

—To assign a specific operational interpretation to these commands.

We will opt for the latter, which results in simpler proofs (but see the discussion in the box of page 38). We start by introducing two distinct program states that we add to the set of state functions: **error** and **blocked**. A program enters the **error** (resp. **blocked**) state whenever an **assert** θ (resp. **assume** θ) statement is executed in a state that *does not satisfy* θ. Whereas the **error** state signals violations of assert statements (a correct program may not enter this state), the **blocked** state is used to eliminate executions that should not be taken into account: when an execution violates an assume statement, then that execution is not required to satisfy any assertions that might be found later. Such an execution will be marked as blocked.

We may now define the natural semantics as follows.

Definition 17 Natural semantics. *The evaluation relation for our iteration-free programming language with assert and assume is defined as the smallest relation*

$$\rightsquigarrow \subseteq \ \mathbf{Comm} \times \Sigma \times (\Sigma \cup \{\textbf{\textit{error}}, \textbf{\textit{blocked}}\})$$

satisfying the following set of rules:

(1) $\langle \mathbf{skip}, s\rangle \rightsquigarrow s$.

(2) $\langle x := e, s\rangle \rightsquigarrow s[x \mapsto [\![e]\!](s)]$.

(3) $\langle \mathbf{assume}\,\theta, s\rangle \rightsquigarrow \textbf{\textit{blocked}}$ *if* $s \not\models \theta$.

(4) $\langle \mathbf{assume}\,\theta, s\rangle \rightsquigarrow s$ *if* $s \models \theta$.

(5) *If* $\langle C_1, s\rangle \rightsquigarrow \textbf{\textit{blocked}}$, *then* $\langle C_1\,;\,C_2, s\rangle \rightsquigarrow \textbf{\textit{blocked}}$.

(6) *If* $\langle C_1, s\rangle \rightsquigarrow \textbf{\textit{error}}$, *then* $\langle C_1\,;\,C_2, s\rangle \rightsquigarrow \textbf{\textit{error}}$.

(7) *If* $\langle C_1, s\rangle \rightsquigarrow s'$, $s' \neq \textbf{\textit{error}}$, $s' \neq \textbf{\textit{blocked}}$, *and* $\langle C_2, s'\rangle \rightsquigarrow s''$, *then* $\langle C_1\,;\,C_2, s\rangle \rightsquigarrow s''$.

(8) *If* $[\![b]\!](s) = \mathbf{T}$ *and* $\langle C^t, s\rangle \rightsquigarrow s'$, *then* $\langle \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f, s\rangle \rightsquigarrow s'$.

(9) *If* $[\![b]\!](s) = \mathbf{F}$ *and* $\langle C^f, s\rangle \rightsquigarrow s'$, *then* $\langle \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f, s\rangle \rightsquigarrow s'$.

(10) *If* $[\![b]\!](s) = \mathbf{F}$, *then* $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s\rangle \rightsquigarrow s$.

(11) *If* $[\![b]\!](s) = \mathbf{T}$ *and* $\langle C, s\rangle \rightsquigarrow \textbf{\textit{blocked}}$ *then* $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s\rangle \rightsquigarrow \textbf{\textit{blocked}}$.

(12) *If* $[\![b]\!](s) = \mathbf{T}$ *and* $\langle C, s\rangle \rightsquigarrow \textbf{\textit{error}}$ *then* $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s\rangle \rightsquigarrow \textbf{\textit{error}}$.

(13) *If* $[\![b]\!](s) = \mathbf{T}$, $\langle C, s\rangle \rightsquigarrow s'$, $s' \neq \textbf{\textit{error}}$, $s' \neq \textbf{\textit{blocked}}$, *and* $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s'\rangle \rightsquigarrow s''$, *then* $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s\rangle \rightsquigarrow s''$.

(14) $\langle \mathbf{assert}\,\theta, s\rangle \rightsquigarrow \textbf{\textit{error}}$ *if* $s \not\models \theta$.

(15) $\langle \mathbf{assert}\,\theta, s\rangle \rightsquigarrow s$ *if* $s \models \theta$.

Observe how rules 3 and 14 signal when an error occurs or the program blocks, and rules 5 and 11, and 6 and 12 respectively, then stop execution (if further statements exist). Rule 7 allows execution to proceed from normal states, and rules 8 and 9 test

to be used as reference, and an axiomatic semantics that will in turn be used for generating verification conditions.

the logical value of a condition and then select the appropriate branch for execution. Rules 10 and 13 handle loop exit (when the condition fails) and iteration (when the loop body is executed normally).

We remark that there are four possible outcomes for execution of a program from a given initial state: the program stops in the **error** state, or it stops in the **blocked** state, or it terminates normally, or it doesn't terminate. This outcome is *unique* for a given initial state (the semantics is deterministic).

We are now in a position to introduce the classic notion of *Hoare triple*, as a syntactic entity to express correctness with respect to a given specification.

3.2 Hoare Triples

A Hoare triple is written as

$$\{\phi\}\, C\, \{\psi\}$$

where the assertions ϕ and ψ are respectively a *precondition* and a *postcondition* for the program C. The triple is interpreted as either true or false in a given state, the idea being that it is interpreted as true exactly when the execution of the program in that state conforms to its specification.

The program assertions used as preconditions and postconditions (and later also as loop invariants) are usually considered to be formulas of first-order logic. The assertion language may range from the one obtained by simply adding existential and universal quantifiers to Boolean expressions, to a much richer language containing, say, functions and predicates together with an axiomatization provided by the user. The assertion language used in [Almeida et al. 2011] is

$$\begin{aligned}\mathbf{Assert} \ni \phi, \psi, \theta \quad ::= \quad & t^{\mathbf{bool}} \mid \top \mid \bot \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \to \psi \mid \phi \leftrightarrow \psi \mid \\ & \forall\, x.\, \phi \mid \exists\, x.\, \phi\end{aligned}$$

where $t^{\mathbf{bool}}$ may extend Boolean expressions with a language provided by the user.

At this point we need to step back and reflect on what a specification is. In the study of the semantics of a language without assert or assume commands, the specification consists exclusively of the precondition and the postcondition, and the program is correct iff whenever it is executed in a state satisfying the precondition, the postcondition will be satisfied in the final state. We now need to adapt this idea to the scope of programs containing asserts and assumes. These are programs that already contain their own built-in specifications, conferred by the assume and assert statements, so our interpretation of Hoare triples must handle both this 'internal' specification and the 'external' specification given by a precondition and a postcondition.

In order to give a formal interpretation of Hoare triples, we now need to express exactly what the meaning of assume and assert should be. Recall that an assert statement contains a formula that must be true when execution goes through it; and an assume *excludes* certain executions from having to conform to asserts that might be subsequently met, as well as to the postcondition. Since the operational semantics stops these executions in the **blocked** state, it suffices to state that executions that do not block also do not enter the **error** state (because of a failed assert), and moreover satisfy the postcondition.

More details on the formal setting

Our formalization here is as simple as possible, and omits some details. First of all, our programming language contains at least one type of expressions, Boolean expressions, which are used in conditional statements (and later in loops) to decide control flow. But of course practical languages will have scope for expressions of at least one more type, and each of these types will have its own domain of interpretation (for instance Boolean expressions are interpreted in $\{\mathbf{T}, \mathbf{F}\}$ and integer expressions over $\mathbf{Z}$). As such, the set Σ of program states is really a *generic function space* rather than a set of functions.

Also, since the evaluation relation $\rightsquigarrow$ depends on the interpretation of expressions, which is being left unspecified, it should in fact be written $\rightsquigarrow_{\mathcal{M}}$, where $\mathcal{M} = (D, I)$ is an *interpretation structure* for the vocabulary describing the concrete syntax of program expressions. This structure provides an interpretation domain D as well as a concrete interpretation of constants and operators, given by I. Similarly, $[\![b]\!]$ and $[\![e]\!]$ should in fact be $[\![b]\!]_{\mathcal{M}}$ and $[\![e]\!]_{\mathcal{M}}$, and finally note that $s \models \theta$ is really shorthand for $[\![\theta]\!]_{\mathcal{M}}(s) = \mathbf{T}$.

Formalizing the semantics of a program logic is a bit tricky because program expressions and first-order logic assertions have to be uniformly interpreted in the same setting. Full details of how this is done can be found in [Loeckx and Sieber 1987]. We follow loosely the detailed formalization given in [Almeida et al. 2011], which considers a global vocabulary that is the union of two distinct vocabularies, one for program expressions, and another for user terms (containing user-provided functions and predicates). The interpretation structures $\mathcal{M}$ for the global vocabulary handle both components. The language of assertions considered in [Almeida et al. 2011] contains

$$\mathbf{Assert} \ni \phi, \psi, \theta \quad ::= \quad t^{\mathbf{bool}} \mid \ldots$$

where $t^{\mathbf{bool}}$ corresponds to Boolean terms constructed over the global vocabulary, i.e. formulas that extend Boolean program expressions with user-provided elements.

The interpretation of the Hoare triple $\{\phi\}\, C\, \{\psi\}$ relies on the operational interpretation of C as well as on the interpretation of the assertions ϕ and ψ. For the interpretation of assertions, given an interpretation structure $\mathcal{M} = (D, I)$ for the global vocabulary, we simply use program states as *assignments* for the domain D, and interpret first-order logic formulas in the standard way. This overlapping between program states and assignments is crucial for the interpretation of Hoare triples. Finally, note that $[\![\phi]\!]$, $[\![\psi]\!]$ would be written more accurately as $[\![\phi]\!]_{\mathcal{M}}$ and $[\![\psi]\!]_{\mathcal{M}}$, and similarly $[\![\{\phi\}\, C\, \{\psi\}]\!]$ as $[\![\{\phi\}\, C\, \{\psi\}]\!]_{\mathcal{M}}$ and $\models \{\phi\}\, C\, \{\psi\}$ as $\models_{\mathcal{M}} \{\phi\}\, C\, \{\psi\}$, to make explicit the parameterization with the interpretation structure.

Definition 18 Interpretation of Hoare triple. *Given $C \in$ **Comm**, $\phi, \psi \in$ **Assert**, and a state $s \in \Sigma$, the Hoare triple $\{\phi\}\, C\, \{\psi\}$ is interpreted in s as either* $\mathbf{T}$ *or* $\mathbf{F}$ *as follows:*

$$[\![\{\phi\}\, C\, \{\psi\}]\!](s) = \mathbf{T} \quad \textit{iff}$$

*if $[\![\phi]\!](s) = \mathbf{T}$ and $\langle C, s\rangle \rightsquigarrow s'$ with $s' \neq$ **blocked**, then $s' \neq$ **error** and $[\![\psi]\!](s') = \mathbf{T}$*

An alternative formulation is the following (notice that **blocked** and **error** are

Boolean expressions vs. first-order assertions

Our setting here is consistent with the fact that in runtime verification and software model checking tools (which typically rely on assume and assert statements for specifying properties) assertions are usually Boolean program expressions. Deductive verification tools on the other hand (where preconditions and postconditions are typically used as part of *contracts*) typically allow for the use of first-order logic formulas as assertions.

In our setting it may well be defensible to restrict all program assertions to Boolean expressions for the sake of practicality, since it is well-known that deciding the validity of quantified formulas is much harder than for propositional formulas. Our setting does not *require* first-order assertions, and in fact, if we look at the present section as simply introducing the formalism required for a proof of soundness of the VC generation mechanisms of Section 2, such assertions are not necessary at all (since preconditions and postconditions are not present in that section). Nonetheless, we choose to have first-order assertions to keep to the spirit of deductive verification formalisms and tools.

What if we ask the converse question? Since we are discussing here essentially mechanisms that produce sound sets of verification conditions, not the feasibility of actually discharging these conditions, why not allow for assume and assert statements to work with full first-order program assertions? After all, asserts (resp. assumes) generalize post-conditions (resp. preconditions), and it may be helpful (for instance while debugging programs) to check at an arbitrary point of the program some property that cannot be expressed directly as a Boolean expression. There is a problem, having to do with the very different ways in which we have formalized assume/assert statements and pre/postconditions. Whereas the latter were interpreted when giving the semantics of Hoare triples, the former were interpreted operationally, which means that for instance the execution of the statement $\mathbf{assert}\,\theta$ forces the interpretation of θ at runtime, and of course, if θ is now a first-order assertion, this is not feasible.

There are two ways out of this situation: the first is to treat assume and assert statements only at the level of the interpretation of Hoare triples (making them behave operationally like **skip**). This is not straightforward, since it requires considering executions 'globally' to express that, whenever an $\mathbf{assert}\,\theta$ statement is executed, if none of the *previously executed* assume statements have failed (i.e. this is an execution that has to be taken into account because it has not been 'blocked' by an assume), then the assertion θ must be satisfied in the current state. This approach will likely lead to more difficult soundness and completeness proofs. The alternative is to simply ignore the issue: while it is true that it is awkward to have first-order assertions in the operational semantics, it can be argued that this semantics is being used just as a reference for the verification of program properties, with first-order assertion checking at runtime appearing as a conceptual device, not as an execution mechanism.

It may have occurred to the reader that in the presence of first-order assertions the precondition and postcondition in a Hoare triple may be internalized into the program by using an initial assume and a final assert statements, resulting in an alternative, simplified interpretation of triples: $[\![\{\phi\}\,C\,\{\psi\}]\!](s) = \mathbf{T}$ iff whenever $\langle \mathbf{assume}\,\phi\,;\,C\,;\,\mathbf{assert}\,\psi, s\rangle \rightsquigarrow s'$, then $s' \neq \mathbf{error}$.

Partial Correctness and Total Correctness

The notion of correctness used in Definition 18 corresponds to *partial correctness*, since it does not require execution to terminate; it simply states that *if* execution terminates, the postcondition must hold. A *total correctness* interpretation, on the other hand, would force termination. Of course in the present context of iteration-free programs the two notions coincide. But it is important to stress that since we have included assume and assert statements in our programming language, our interpretation of Hoare triples contains a design decision, which is to forbid assert statements to fail as part of the partial correction interpretation: programs are not forced to terminate, but if they do it can't be in the **error** state.

Both notions of partial and total correctness are also used in the guarded commands / predicate transformers setting, and it is important to clarify a difference between our formalization and the one used by that community, for whom partial correctness does not imply that programs do not terminate in the error state – only total correctness implies that. Partial correctness is instead understood as "if no assert fails, then the postcondition must be satisfied in the final state". The two notions of correctness correspond to different predicate transformers: the notion of weakest precondition introduced in Section 2.3 is a total correctness notion, since it describes states that result in executions in which no assert fails. The weaker notion of *weakest liberal precondition* on the other hand is a partial correctness notion.

distinct states):

$$\text{if } [\![\phi]\!](s) = \mathbf{T} \text{ and } \langle C, s\rangle \rightsquigarrow s', \text{ then } s' \neq \mathbf{error}, \text{ and if } s' \neq \mathbf{blocked} \text{ then } [\![\psi]\!](s') = \mathbf{T}$$

As would be expected, a Hoare triple is valid if it is true in every state:

DEFINITION 19 VALID HOARE TRIPLE. *The triple* $\{\phi\}\, C\, \{\psi\}$ *is* valid*, written* $\models \{\phi\}\, C\, \{\psi\}$*, iff* $[\![\{\phi\}\, C\, \{\psi\}]\!](s) = \mathbf{T}$ *for every state* $s \in \Sigma$.

In this case the program C is said to be *correct* with respect to the specification consisting of the precondition ϕ and the postcondition ψ.

3.3 Axiomatic Semantics – System H

The interpretation of a Hoare triple was given in operational terms. The *truth* of a Hoare triple in a given state means that the execution of the program in that state will not violate the specification. It is reminiscent of *dynamic verification* (also known as *runtime verification*), a technique that monitors each execution of the program and signals violations that may occur, based on a special compiler that introduces monitoring code into the executable program. This verification technique can be used as a complement to the standard testing and debugging mechanisms that are used for validating programs.

Now in order to establish the *validity* of a triple, i.e. to ensure that *every* execution is safe, we could try to run the program starting from every possible state (exhaustive testing), which would of course not be practical. Alternatively we can resort to formal verification, as explained in Section 2, to reason about correctness statically, without executing the program. In order to do this we will employ a *deductive system* that will allow us to derive certain Hoare triples from other triples, in the structured manner that is typical of logical inference systems. Hoare

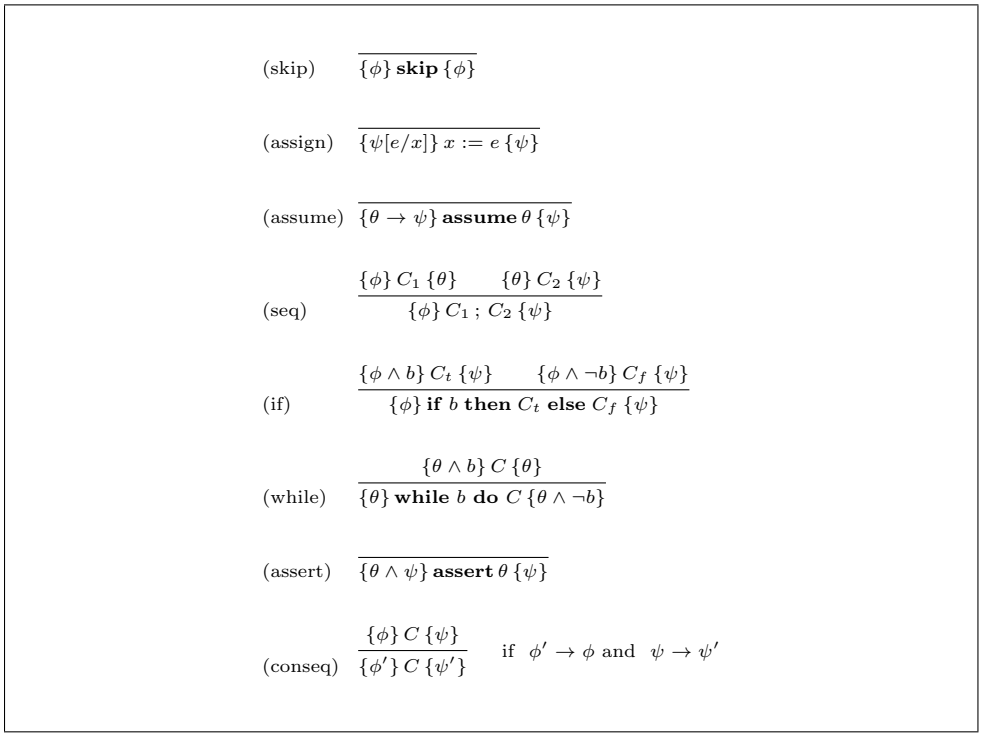

Fig. 3. Inference system of Hoare logic – System H

introduced such a system for While programs in 1969 [Hoare 1969]. We will here call this inference system, to which we have added rules for the assume and assert commands, *system* H. It is shown in Figure 3.

System H admits multiple derivations for the same Hoare triple, and does not force a particular strategy for constructing them (in particular note that the sequence rule allows assertions to be propagated both forward and backward). However, the crucial (assign) rule is based on a weakest precondition calculation, and as such, derivations based on backward propagation are in a sense more natural in this system. The new (assume) and (assert) rules follow the assignment rule in this respect: they propagate the postcondition ψ backward, according to the definition of the weakest precondition predicate transformer for the guarded commands language (see Section 2.3).

Logics for reasoning about program correctness are known as *program logics*; Hoare logic is the foremost example of such a logic. System H is the proof system of Hoare logic, whereas its semantics is given by Definition 19. As is the case in any logic, the proof system can only be useful if it enjoys suitable *soundness* and *completeness* properties with respect to the semantics.

System H is *sound*: it does not derive triples that are not valid according to Definition 19. It is easy to show that extending the original logic with our rules for the assume and assert commands preserves soundness:

PROPOSITION 20 SOUNDNESS OF SYSTEM H. *Let* $C \in \mathbf{Comm}$ *and* $\phi, \psi \in \mathbf{Assert}$ *such that* $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$. *Then* $\models \{\phi\}\, C\, \{\psi\}$.

PROOF. By induction on the structure of the derivation. We detail here just the assume and assert base cases, since these commands are not part of the While language used in standard Hoare logic.

C is $\mathbf{assume}\,\theta$*..* If $[\![\theta \to \psi]\!](s) = \mathbf{T}$ and $\langle \mathbf{assume}\,\theta, s\rangle \rightsquigarrow s'$ with $s' \neq \mathbf{blocked}$, this implies by the operational semantics that $s \models \theta$ and $s' = s$, thus $s' \neq \mathbf{error}$, and indeed $[\![\psi]\!](s') = \mathbf{T}$.

C is $\mathbf{assert}\,\theta$*..* If $[\![\theta \wedge \psi]\!](s) = \mathbf{T}$, then $[\![\theta]\!](s) = \mathbf{T}$ and thus by the operational semantics $\langle \mathbf{assert}\,\theta, s\rangle \rightsquigarrow s'$ with $s' = s$, thus $s' \neq \mathbf{error}$, and moreover $[\![\psi]\!](s') = \mathbf{T}$.

□

The issue of completeness of Hoare logic on the other hand deserves a deeper discussion. The account below is based on [Apt 1981]. First of all, it is clear that only a *relative* notion of completeness can be achieved, for the simple reason that the application of the consequence rule is guarded by assertions. System H does not contain any rules for deriving these assertions, and moreover it would likely not be possible to extend the system with an adequate axiomatization, since interesting assertion languages may themselves be incomplete (it is difficult to imagine reasoning about programs without requiring arithmetics, for instance). Cook [Cook 1978] notes that restricting the specification language (i.e. the language of assertions occurring in preconditions and postconditions) does not solve the problem, as shown by triples of the form $\{\top\}\, C\, \{\bot\}$, expressing that no execution of C halts. If the programming language is such that the halting problem is undecidable, the validity of such a triple does not imply that it can be derived in system H, which again confirms the incompleteness of the system.

The solution to this first problem is easy. Recall from the box of page 37 that the interpretation of assertions and Hoare triples is implicitly indexed by a model $\mathcal{M}$. We will equally assume that derivation of Hoare triples in system H is indexed by the *complete theory* of this structure (the set of all valid assertions under $\mathcal{M}$), so $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$ becomes shorthand for $\mathrm{Th}(\mathcal{M}) \vdash_{\mathsf{H}} \{\phi\}\, \mathrm{C}\, \{\psi\}$, which means that when constructing derivations in system H one simply checks, when applying the guarded (conseq) rule, whether the side conditions are elements of $\mathrm{Th}(\mathcal{M})$.[5]

As to the second problem, rather than explaining it now we will formulate the completeness result in a restricted way that already solves it. The problem will manifest itself as we write the proof of completeness; we will comment on it then. Let $C \in \mathbf{Comm}$ and $\phi \in \mathbf{Assert}$, and denote by $\mathrm{post}(\phi, \mathrm{C})$ the set of states $\{s' \in \Sigma \mid \langle C, s\rangle \rightsquigarrow s'$ for some $s \in \Sigma$ such that $[\![\phi]\!](s) = \mathbf{T}\}$.

[5]In fact this should not be very surprising with our presentation of the (conseq) rule: the fact that the first-order guards appear as side conditions indicates that they are not supposed to be derived within the inference system. It is not unusual to see the guards presented as premisses of the rule, but strictly speaking the resulting systems are incomplete. Note also that the soundness result could also be formulated making explicit the sets of formulas used as side conditions in the derivation, but in that result it is not critical whether these assertions are derived in a formal system, or checked to be members of the theory of $\mathcal{M}$.

Definition 21 Expressiveness. *The assertion language* **Assert** *is said to be* expressive *with respect to the command language* **Comm** *and interpretation structure* $\mathcal{M}$*, if for every* $\phi \in$ **Assert** *and* $C \in$ **Comm** *there exists* $\psi \in$ **Assert** *such that* $s \models \psi$ *iff* $s \in \mathrm{post}(\phi, \mathrm{C})$ *for any* $s \in \Sigma$*.*

ψ is said to be the *postcondition* of C with respect to ϕ; the expressiveness condition guarantees its existence.

Proposition 22 Completeness of System H in the sense of Cook. *Let* $C \in$ **Comm**, $\phi, \psi \in$ **Assert**, *and* $\mathcal{M}$ *be a structure such that* **Assert** *is expressive with respect to* **Comm** *and* $\mathcal{M}$*. If* $\models_{\mathcal{M}} \{\phi\}\, C\, \{\psi\}$ *then* $\mathrm{Th}(\mathcal{M}) \vdash_{\mathsf{H}} \{\phi\}\, \mathrm{C}\, \{\psi\}$.

Proof. The proof proceeds by induction on the structure of the program C. In what follows we write $[\![\cdot]\!]$ for $[\![\cdot]\!]_{\mathcal{M}}$, $\models$ for $\models_{\mathcal{M}}$, and $\vdash_{\mathsf{H}}$ for $\mathrm{Th}(\mathcal{M}) \vdash_{\mathsf{H}}$. We give here the base cases, in particular *assume* and *assert* that are not part of standard Hoare logic. We also show the sequence case, in order to illustrate the need for the expressiveness assumption. The reader is directed to [Apt 1981] for the more elaborate case of *while* (which in fact is easier to prove if the expressiveness requirement is stated in terms of the definability of weakest preconditions rather than strongest postconditions).

C is $x := e$.. We have that for all states s and s', if $[\![\phi]\!](s) = \mathbf{T}$ and $\langle x := e, s\rangle \rightsquigarrow s'$ with $s' \neq$ **blocked**, then $s' \neq$ **error** and $[\![\psi]\!](s') = \mathbf{T}$. Now since $s' = s[x \mapsto [\![e]\!](s)]$, this assumption can be rewritten as: if $[\![\phi]\!](s) = \mathbf{T}$ then $[\![\psi]\!](s[x \mapsto [\![e]\!](s)]) = \mathbf{T}$. But then also $[\![\psi[e/x]]\!](s) = \mathbf{T}$, and thus $\models \phi \to \psi[e/x]$. The derivation $\vdash_{\mathsf{H}} \{\phi\}\, x := e\, \{\psi\}$ follows from the (assign) and (conseq) rules of system H.

C is **assume** θ*.*. We have that for all states s, s', if $[\![\phi]\!](s) = \mathbf{T}$ and $\langle \mathbf{assume}\, \theta, s\rangle \rightsquigarrow s'$ with $s' \neq$ **blocked**, then $s' \neq$ **error** and $[\![\psi]\!](s') = \mathbf{T}$. Now note that the assume does not block exactly when $[\![\theta]\!](s) = \mathbf{T}$, in which case $s' = s$. Thus we can rephrase our assumption as: for every state s, if $[\![\phi]\!](s) = \mathbf{T}$ and $[\![\theta]\!](s) = \mathbf{T}$, then $[\![\psi]\!](s) = \mathbf{T}$. This implies that $[\![\phi \wedge \theta \to \psi]\!](s) = \mathbf{T}$ for every s, and thus $\models \phi \to \theta \to \psi$. Thus $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$ follows from the (assume) rule of system H and (conseq).

C is **assert** θ*.*. We have that for all states s and s', if $[\![\phi]\!](s) = \mathbf{T}$ and $\langle \mathbf{assert}\, \theta, s\rangle \rightsquigarrow s'$ with $s' \neq$ **blocked**, then $s' \neq$ **error** and $[\![\psi]\!](s') = \mathbf{T}$. Note that assert statements never block, and do not fail (produce error) exactly when $[\![\theta]\!](s) = \mathbf{T}$, in which case $s' = s$. Thus we can rephrase the assumption as: for every state s, if $[\![\phi]\!](s) = \mathbf{T}$ then $[\![\theta]\!](s) = \mathbf{T}$ and $[\![\psi]\!](s) = \mathbf{T}$. This implies that $[\![\phi \to \theta \wedge \psi]\!](s) = \mathbf{T}$ for every s, and thus $\models \phi \to \theta \wedge \psi$. Then $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$ follows from the (assert) rule of system H and (conseq).

C is $C_1\,;\, C_2$.. We assume that for all states s and s'', if $[\![\phi]\!](s) = \mathbf{T}$ and $\langle C_1\,;\, C_2, s\rangle \rightsquigarrow s''$ with $s'' \neq$ **blocked**, then $s'' \neq$ **error** and $[\![\psi]\!](s'') = \mathbf{T}$. We have the following induction hypotheses, where we fix the formulas ϕ and ψ :

IH1:. For every formula θ, if $\models \{\phi\}\, C_1\, \{\theta\}$ then $\vdash_{\mathsf{H}} \{\phi\}\, C_1\, \{\theta\}$.

IH1:. For every formula θ, if $\models \{\theta\}\, C_2\, \{\psi\}$ then $\vdash_{\mathsf{H}} \{\theta\}\, C_2\, \{\psi\}$.

Now it is apparently straightforward to conclude this proof case by applying both induction hypotheses. But in fact care is required. Assuming $C_1\,;\, C_2$ does not block (and thus neither do C_1 or C_2), we know it doesn't fail, and moreover ψ is satisfied

in the final state. But the assumptions do not immediately indicate a concrete intermediate assertion θ. It is here that Cook's expressiveness requirement becomes useful: we know there exists $\Theta \in \mathbf{Assert}$ such that $s \models \Theta$ iff $s \in \mathrm{post}(\phi, \mathrm{C}_1)$ for any $s \in \Sigma$. So we immediately have $\models \{\phi\}\, C_1\, \{\Theta\}$, and since $\models \{\phi\}\, C_1 ; C_2\, \{\psi\}$ we also necessarily have that $\models \{\Theta\}\, C_2\, \{\psi\}$. Both induction hypotheses then apply with Θ for θ, which allows for the proof case to be concluded using the (seq) rule of system H.

□

As a final remark, we note that completeness means in particular that (under the expressiveness assumption) it is always possible to write appropriate invariants for a program to be deductively shown correct with respect to a specification, if indeed it is correct with respect to it.

3.4 Goal-directed Program Logic

The consequence rule of Hoare logic plays two roles in the construction of derivations: on one hand it is used to provide the 'glue' that allows the remaining rules to be plugged together, when the formulas do not match because they are either too strong or too weak. On the other hand, it can be applied to reuse parts of proofs: for instance if the code contains some form of sub-routine, then instead of producing one different proof for each call of such a routine, it is a good idea to produce a single one with respect to a sufficiently rich specification (usually called a *contract*). This same proof may then be reused, plugged into different places in the overall derivation, precisely by using the consequence rule to adapt it to the local requirements of each call.

The second role of (conseq) described above can alternatively be performed by a dedicated rule for procedure calls, and the first role does not in fact require a dedicated rule: the remaining rules can be modified to perform that task themselves. The problem with the consequence rule is that it is a source of ambiguity in the construction of derivations, since it is the only rule in system H whose application is not directed by the structure of the program, and may thus occur at any point. In [Frade and Pinto 2011] we have introduced a *goal-directed* version of Hoare logic, without consequence rule, that is free of this ambiguity. This is important for the purpose of generating verification conditions, since this should be done by a simple algorithm that does not have to choose between different rules to apply.

But from the point of view of the generation of verification conditions there is another aspect that has to be taken into account: a VCGen algorithm should not have to make guesses regarding loop invariants. In all popular deductive verification tools, these are the responsibility of the user and must be provided as inputs. Henceforth we will thus consider that loop invariants are provided as *annotations* that are part of the code: they play no role in the semantic behavior of the programs, but are used for generating verification conditions. The goal-directed system of Hoare logic handles triples containing such invariant-annotated programs.

We remark that soundness is not much affected by this reformulation: if the VCs generated from a given annotated Hoare triple (i.e. a triple in which all the loops in the program are annotated with invariants) are valid, then it is possible to produce

a derivation using the annotated invariants, and the triple is valid.

The completeness result requires more consideration. In fact, an annotated program may well be correct with respect to its specification, and yet contain annotations that will not allow us to prove its correctness, because the suggested invariants may not be invariants at all, or else they may be invariants that are too weak to allow a given postcondition to be established, in which case it will not be possible to construct a derivation in the program logic. Instead of trying to obtain a general completeness result, we will instead introduce a notion of *correctly-annotated program*, and focus on the completeness of VC generation for these programs.

We start by giving syntax for annotated programs as follows, where the θ occurring in loop bodies are program annotations (formulas to be used as loop invariants).

$$\begin{array}{rcl} \mathbf{AComm} \ni C & ::= & \mathbf{skip} \mid C\,;\, C \mid x := e \mid \mathbf{assume}\,\theta \mid \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \\ & & \mathbf{while}\ b\ \mathbf{do}\,\{\theta\}\,C \mid \mathbf{assert}\,\theta \end{array}$$

It will be useful to have a function that erases all annotations from a program:

DEFINITION 23. *The function* $\lfloor\cdot\rfloor : \mathbf{AComm} \to \mathbf{Comm}$ *is defined as follows:*

$$\begin{aligned} \lfloor \mathbf{skip} \rfloor &= \mathbf{skip} \\ \lfloor x := e \rfloor &= x := e \\ \lfloor \mathbf{assume}\,\theta \rfloor &= \mathbf{assume}\,\theta \\ \lfloor C_1\,;\, C_2 \rfloor &= \lfloor C_1 \rfloor\,;\, \lfloor C_2 \rfloor \\ \lfloor \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C_f \rfloor &= \mathbf{if}\ b\ \mathbf{then}\ \lfloor C^t \rfloor\ \mathbf{else}\ \lfloor C_f \rfloor \\ \lfloor \mathbf{while}\ b\ \mathbf{do}\,\{\theta\}\,C \rfloor &= \mathbf{while}\ b\ \mathbf{do}\ \lfloor C \rfloor \\ \lfloor \mathbf{assert}\,\theta \rfloor &= \mathbf{assert}\,\theta \end{aligned}$$

While system H considers standard, non-annotated programs, the goal-directed system Hg targets specifically Hoare triples containing annotated programs. Its rules are given in Figure 4. It is clear that the (while) rule uses the annotations in the loops, leaving no room for invariants to be invented.

The soundness result given in [Frade and Pinto 2011] extends in a straightforward way to our language with assume and assert statements:

PROPOSITION 24 SOUNDNESS OF SYSTEM Hg. *Let* $C \in \mathbf{AComm}$ *and* $\phi, \psi \in \mathbf{Assert}$.

$$\textit{If} \qquad \vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\} \qquad \textit{then} \qquad \vdash_{\mathsf{H}} \{\phi\}\,\lfloor C \rfloor\,\{\psi\}$$

PROOF. *By induction on the derivation of* $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$ □

As explained before, the reverse implication does not hold, since there may well be valid triples that are impossible to derive since they contain wrong annotations. But it does hold for triples not containing loops. To show this, we require the following lemma that states the admissibility of the consequence rule in Hg.

LEMMA 25. *Let* $C \in \mathbf{AComm}$ *and* $\phi, \psi, \phi', \psi' \in \mathbf{Assert}$ *such that* $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$, $\models \phi' \to \phi$, *and* $\models \psi \to \psi'$. *Then* $\vdash_{\mathsf{Hg}} \{\phi'\}\,C\,\{\psi'\}$.

PROPOSITION 26. *Let* $C \in \mathbf{AComm}$ *containing no loop statements, and* $\phi, \psi \in \mathbf{Assert}$.

$$\textit{If} \qquad \vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi\} \qquad \textit{then} \qquad \vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$$

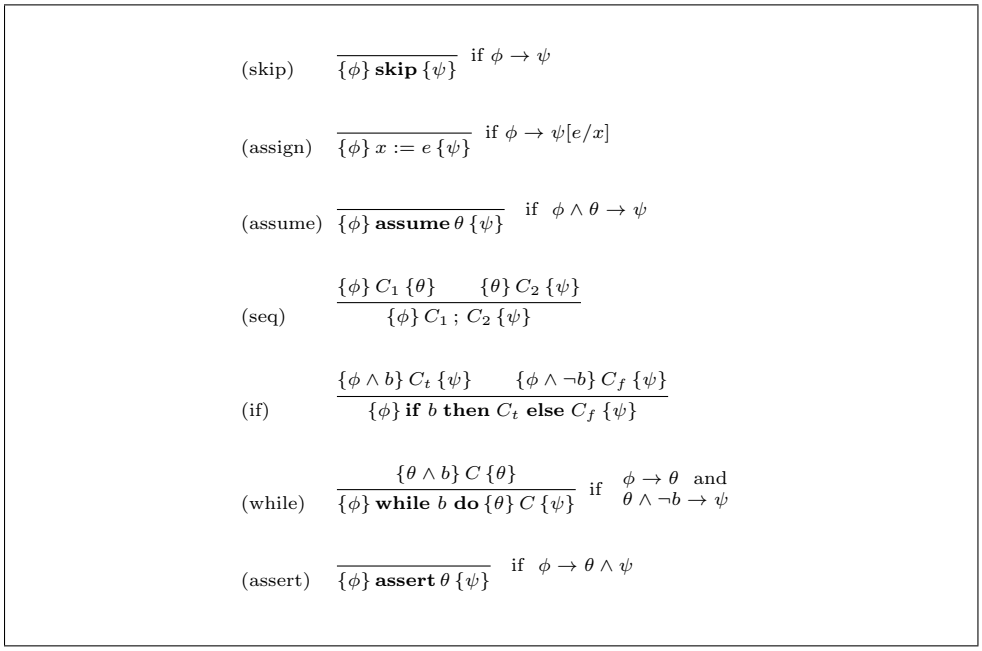

Fig. 4. Inference system of Hoare logic without consequence rule: system Hg

PROOF. *By induction on the derivation of* $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi\}$*, using Lemma 25.* □

For programs with loops, we note that if a triple $\{\phi\}\,C\,\{\psi\}$ is derivable in H, then it is certain that *there exists an annotated version* C' of C such that $\{\phi\}\,C'\,\{\psi\}$ is derivable in Hg (it suffices to use the invariants that were used in the derivation of the non-annotated triple). So instead of working with a completeness result we simply restrict our attention to such correctly-annotated versions of programs.

DEFINITION 27 CORRECTLY-ANNOTATED PROGRAM. *Let* $C \in \mathbf{AComm}$ *and* $\phi, \psi \in \mathbf{Assert}$*.* C *is* correctly-annotated *with respect to* (ϕ, ψ) *if* $\vdash_{\mathsf{H}} \{\phi\}\,\lfloor C \rfloor\,\{\psi\}$ *implies* $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$*.*

Note that the loop invariants annotated in a program may well depend on the specification of the program, since their purpose is to allow for the program to be proved correct with respect to it.

Derivations in system Hg are quite restricted: since every conclusion triple has arbitrary preconditions and postconditions, and side conditions are distributed among various rules of the system, the structure of the programs determines uniquely the rules to be applied in the construction of derivations. Moreover, the annotations contain the loop invariants to be used in the (while) rule, which means that the shape of derivations and the invariants used in them are fixed.

In spite of the above, different derivations can still be constructed for the same Hoare triple, differing only in terms of the intermediate formulas used. System Hg is agnostic with respect to a strategy for choosing these assertions; two straightforward strategies are

Forward propagation.. Propagate conditions forward in the (seq) rule; use axioms to determine postconditions that make the side conditions hold trivially for the given preconditions.

Backward propagation.. Propagate conditions backward in the (seq) rule; use axioms to determine preconditions that make the side conditions hold trivially for the given postconditions.

We will end the present section with a VCGen algorithm derived from system Hg, based on the backward propagation strategy. In Section 4 we will introduce a program logic specifically for the single-assignment branching programs of Section 2, and derive an efficient forward propagation VCGen for it. System Hg (restricted to branching programs) will play an important role in the proof of completeness of that logic.

3.5 A Backward Propagation VCGen

The idea behind a verification conditions generator is that it is possible to write a function that, following the structure of the program, produces a set of assertions corresponding exactly to the side conditions of one particular derivation in the inference system (in this case a Hg derivation). We note the following:

—The VCGen imposes a strategy to obtain one particular derivation among the set of possible derivations for the same triple.

—The side conditions of this derivation can be collected by a recursive traversal of the program, without actually constructing the explicit derivation.

—Usually, the set of side conditions is entirely calculated before any validity checking takes place. What this means is that the VCGen considers one of the *candidate derivations* of the triple and then checks the validity of its side conditions to see if the candidate is actually a derivation.

—But then the following question arises: if this particular candidate derivation is *not* a derivation (because one of the side conditions is not valid), does this mean that the triple is not valid? Or just that the chosen strategy was not adequate and another one should be attempted?

The answer to the last question is that in fact no other strategies have to be considered: if the set of VCs generated by the backward propagation VCGen is not valid, then the Hoare triple is not valid. In other words, the VCGen is complete.

From the observation of the rules of system Hg it is clear that the only unknown aspect in the recursive calculation of the set of VCs is the intermediate assertion θ in the (seq) rule. Let us then write a function that will give us an intermediate assertion by propagating a given postcondition through a command. The function resembles a weakest precondition calculation, except that it approximates the weakest precondition of a loop by its annotated invariant.

DEFINITION 28. *The weakest precondition approximation function* wprec : $\mathbf{AComm}\times$

Assert $\to$ **Assert** *is defined as follows.*

$$\begin{aligned}
\mathsf{wprec}(\mathbf{skip}, \psi) &= \psi \\
\mathsf{wprec}(x := e, \psi) &= \psi[e/x] \\
\mathsf{wprec}(\mathbf{assume}\,\theta, \psi) &= \theta \;\to\; \psi \\
\mathsf{wprec}(C_1; C_2, \psi) &= \mathsf{wprec}(C_1, \mathsf{wprec}(C_2, \psi)) \\
\mathsf{wprec}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, \psi) &= (b \to \mathsf{wprec}(C_t, \psi)) \wedge (\neg b \to \mathsf{wprec}(C_f, \psi)) \\
\mathsf{wprec}(\mathbf{while}\ b\ \mathbf{do}\,\{\theta\}\,C, \psi) &= \theta \\
\mathsf{wprec}(\mathbf{assert}\,\theta, \psi) &= \theta \;\wedge\; \psi
\end{aligned}$$

It is now straightforward to write the VCGen function, using wprec to compute the intermediate assertions.

Definition 29 Backward-propagation VCGen. *The verification conditions generator function* $\mathsf{VC} : \mathbf{Assert} \times \mathbf{AComm} \times \mathbf{Assert} \to \mathcal{P}(\mathbf{Assert})$ *is defined as follows:*

$$\begin{aligned}
\mathsf{VC}(\phi, \mathbf{skip}, \psi) &= \{\phi \to \psi\} \\
\mathsf{VC}(\phi, x := e, \psi) &= \{\phi \to \psi[e/x]\} \\
\mathsf{VC}(\phi, \mathbf{assume}\,\theta, \psi) &= \{\phi \wedge \theta \to \psi\} \\
\mathsf{VC}(\phi, C_1\,;\, C_2, \psi) &= \mathsf{VC}(\phi, C_1, \mathsf{wprec}(C_2, \psi)) \cup \mathsf{VC}(\mathsf{wprec}(C_2, \psi), C_2, \psi) \\
\mathsf{VC}(\phi, \mathbf{while}\ b\ \mathbf{do}\,\{\theta\}\,C, \psi) &= \{\phi \to \theta, \theta \wedge \neg b \to \psi\} \cup\ \mathsf{VC}(\theta \wedge b, C, \theta) \\
\mathsf{VC}(\phi, \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, \psi) &= \mathsf{VC}(\phi \wedge b, C_t, \psi) \cup \mathsf{VC}(\phi \wedge \neg b, C_f, \psi) \\
\mathsf{VC}(\phi, \mathbf{assert}\,\theta, \psi) &= \{\phi \to \theta \wedge \psi\}
\end{aligned}$$

The VCGen is sound and complete with respect to system Hg (see [Almeida et al. 2011] for proof):

Proposition 30. *Let* $C \in \mathbf{AComm}$ *and* $\phi, \psi \in \mathbf{Assert}$*. Then*

$$\models \mathsf{VC}(\phi, C, \psi) \quad \textit{iff} \quad \vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$$

By composing Propositions 30, 24, and 20 we get the following soundness result for VC generation:

Proposition 31. *Let* $C \in \mathbf{AComm}$ *and* $\phi, \psi \in \mathbf{Assert}$*.*

$$\textit{If} \quad \models \mathsf{VC}(\phi, C, \psi) \quad \textit{then} \quad \models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$$

The corresponding completeness result applies to correctly-annotated programs only. It follows from Proposition 22, Definition 27, and Proposition 30:

Proposition 32. *Let* $C \in \mathbf{AComm}$ *and* $\phi, \psi \in \mathbf{Assert}$ *such that* C *is correctly-annotated with respect to* (ϕ, ψ)*.*

$$\textit{If} \quad \models \{\phi\}\, \lfloor C \rfloor\, \{\psi\} \quad \textit{then} \quad \models \mathsf{VC}(\phi, C, \psi)$$

The above algorithm was easy to write, but is far from optimal. In fact, it generates many tautological verification conditions of the form $\phi \to \phi$. This is due to the fact that in the case of the base commands, the function wprec returns a precondition that makes the side condition trivially valid. As such, it does not need to be included in the set of verification conditions. This also applies to one of the side conditions of the (while) rule.

In fact, the backward propagation implemented by wprec means that the recursive VC function does not even need to include preconditions in its parameters if one adds $\phi \rightarrow \mathsf{wprec}(C, \psi)$ to the set of VCs. With this in mind, the VCGen can be simplified to the following.

$$\begin{aligned}
\mathsf{aux}(\mathbf{skip}, \psi) &= \{\} \\
\mathsf{aux}(x := e, \psi) &= \{\} \\
\mathsf{aux}(\mathbf{assume}\,\theta, \psi) &= \{\} \\
\mathsf{aux}(C_1\,;\, C_2, \psi) &= \mathsf{aux}(C_1, \mathsf{wprec}(C_2, \psi)) \cup \mathsf{aux}(C_2, \psi) \\
\mathsf{aux}(\mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\, C, \psi) &= \{\theta \wedge \neg b \rightarrow \psi\} \cup\ \mathsf{aux}(C, \theta) \\
\mathsf{aux}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, \psi) &= \mathsf{aux}(C_t, \psi) \cup \mathsf{aux}(C_f, \psi) \\
\mathsf{aux}(\mathbf{assert}\,\theta, \psi) &= \{\} \\
\\
\mathsf{VC}(\phi, C, \psi) &= \{\phi \rightarrow \mathsf{wprec}(C, \psi)\} \cup \mathsf{aux}(C, \psi)
\end{aligned}$$

Let us now consider the efficiency of the VCs generated by this VCGen. First, it is clear from Definition 28 that $\mathsf{wprec}(C, \psi)$ contains the formula ψ, except in the case when C is a *while* command. Now take the calculation

$$\mathsf{aux}(C_1\,;\, \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, \psi)$$

The recursive call on C_1 will be $\mathsf{aux}(C_1, \mathsf{wprec}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, \psi))$, or

$$\mathsf{aux}(C_1, (b \rightarrow \mathsf{wprec}(C_t, \psi)) \wedge (\neg b \rightarrow \mathsf{wprec}(C_f, \psi)))$$

We are thus in the presence of the exponential pattern corresponding to path enumeration as described in Section 2, since two copies of the postcondition ψ may be propagated to C_1. This is hardly surprising, since one of the points of using intermediate single-assignment form for generating VCs is precisely the fact that the use of this form makes possible the generation of efficient VCs.

A more detailed presentation of this VCGen can be found in [Frade and Pinto 2011] or [Almeida et al. 2011]. In the next section we will derive one of the VC generation mechanisms examined in Section 2, following the semantically justified method used in the present section.

4. SINGLE-ASSIGNMENT BRANCHING PROGRAMS: LOGIC AND VCGEN

In Section 2 we have hopefully convinced the reader of the equivalence of various verification condition generation mechanisms for branching (non-iterating) code. We havent't proved that any of them is sound or complete, but their equivalence means that the soundness of one of them entails the soundness of all the others, and the same goes for completeness. In Section 3 on the other hand we studied the semantic framework underlying formal program verification, in particular the generation of efficient verification conditions for While programs with assume and assert commands.

We will now see how the semantic tools used in Section 3 can be used to prove soundness and completeness of one of the VCGens for SA code introduced in Section 2. We will define formally the notion of single-assignment branching program,

and then study a program logic and the generation of verification conditions based on this logic.

We will then show how standard (i.e. non-SA) branching code can be verified by first translating it into an intermediate single-assignment form and then verifying the resulting SA code. A major advantage of this technique, which is sound and complete, is that it avoids the exponential explosion problem explained in Section 2 (other advantages are briefly explained in Section 5.3). Although the major existing tools for program verification indeed resort to the use of an intermediate SA form, a formal account of this technique is not, as far as we know, available in the literature.

In the rest of this section whenever we write 'program' we refer to branching (i.e. non-iterating) programs.

4.1 Single-assignment Programs and Hoare Triples

The notion of (dynamic) single-assignment program used in Section 2 can be defined inductively, based on the sets of variables occuring in, and assigned by, a program.

DEFINITION 33. *The sets of variables occurring in a program and variables assigned in a program are defined as follows, where* $\mathsf{Vars}(e)$ *denotes the set of variables occurring in the expression* e*.*

$$
\begin{aligned}
\mathsf{Vars}(\mathbf{skip}) &= \emptyset \\
\mathsf{Vars}(x := e) &= \{x\} \cup \mathsf{Vars}(e) \\
\mathsf{Vars}(\mathbf{assume}\,\theta) &= \mathsf{Vars}(\theta) \\
\mathsf{Vars}(C_1\,;\,C_2) &= \mathsf{Vars}(C_1) \cup \mathsf{Vars}(C_2) \\
\mathsf{Vars}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f) &= \mathsf{Vars}(b) \cup \mathsf{Vars}(C_t) \cup \mathsf{Vars}(C_f) \\
\mathsf{Vars}(\mathbf{assert}\,\theta) &= \mathsf{Vars}(\theta)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Asgn}(\mathbf{skip}) &= \emptyset \\
\mathsf{Asgn}(x := e) &= \{x\} \\
\mathsf{Asgn}(\mathbf{assume}\,\theta) &= \emptyset \\
\mathsf{Asgn}(C_1\,;\,C_2) &= \mathsf{Asgn}(C_1) \cup \mathsf{Asgn}(C_2) \\
\mathsf{Asgn}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f) &= \mathsf{Asgn}(C_t) \cup \mathsf{Asgn}(C_f) \\
\mathsf{Asgn}(\mathbf{assert}\,\theta) &= \emptyset
\end{aligned}
$$

DEFINITION 34 SINGLE-ASSIGNMENT PROGRAM. *The set* $\mathbf{Comm}^{\mathrm{SA}}$ *of Single-Assignment programs is defined as follows:*

—$\mathbf{skip} \in \mathbf{Comm}^{\mathrm{SA}}$.
—$x := e \in \mathbf{Comm}^{\mathrm{SA}}$ *if* $x \notin \mathsf{Vars}(e)$.
—$\mathbf{assume}\,\theta \in \mathbf{Comm}^{\mathrm{SA}}$.
—$C_1\,;\,C_2 \in \mathbf{Comm}^{\mathrm{SA}}$ *if* $C_1, C_2 \in \mathbf{Comm}^{\mathrm{SA}}$, *and* $\mathsf{Vars}(C_1) \cap \mathsf{Asgn}(C_2) = \emptyset$.
—$\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f \in \mathbf{Comm}^{\mathrm{SA}}$ *if* $C_t, C_f \in \mathbf{Comm}^{\mathrm{SA}}$, *and* $\mathsf{Vars}(b) \cap (\mathsf{Asgn}(C_t) \cup \mathsf{Asgn}(C_f)) = \emptyset$.
—$\mathbf{assert}\,\theta \in \mathbf{Comm}^{\mathrm{SA}}$.

The first thing to note when reasoning with triples containing SA programs is that there is not much use in allowing preconditions to contain free occurrences of variables that are assigned in the program, since in the paths containing these assignments the initial values of the variables cannot be read. But Hoare triples containing SA programs result from the translation of standard Hoare triples, and the adequate way to translate triples to SA form will naturally avoid these situations.

As an example of this consider for instance the triple

$$\{x > 0\}\, x := 10\, \{x > 0\}$$

It does not make sense to translate this to SA form as

$$\{x_1 > 0\}\, x_1 := 10\, \{x_1 > 0\}$$

but instead as

$$\{x_0 > 0\}\, x_1 := 10\, \{x_1 > 0\}$$

A way to see this is that the precondition is handled as if it was part of the program: when translating the statement $x := 10$ we consider that x has occurred before in the program, thus a fresh version is required for the assignment.

As a second example consider

$$\{x > 0\}\, \textbf{if } b \textbf{ then } x := 10 \textbf{ else skip}\, \{x > 0\}$$

This seems trickier because one of the branches does not assign to x. Consequently, the x occurring in the postcondition refers to the value assigned to x if the first branch is executed, and to the initial value of x if the second branch is executed. So neither of the following would be an adequate translation:

$$\{x_0 > 0\}\, \textbf{if } b \textbf{ then } x_1 := 10 \textbf{ else skip}\, \{x_0 > 0\}$$

$$\{x_0 > 0\}\, \textbf{if } b \textbf{ then } x_1 := 10 \textbf{ else skip}\, \{x_1 > 0\}$$

But recall that converting programs to dynamic SA form requires the inclusion of code to synchronize version variables from the different branches. In this example this could be done for instance by using a fresh variable x_2:

$$\{x_0 > 0\}\, \textbf{if } b \textbf{ then } x_1 := 10\,;\, x_2 := x_1 \textbf{ else } x_2 := x_0\, \{x_2 > 0\}$$

This discussion justifies the fact that in what follows, results regarding Hoare triples $\{\phi\}\, C\, \{\psi\}$ containing SA programs will in general be subject to the condition that C does not assign variables occurring free in ϕ. We will use the notation $\phi \# C$ to denote this fact, i.e. $\mathsf{Asgn}(C) \cap \mathsf{FV}(\phi) = \emptyset$.

4.2 Program Logic

We will now introduce a specific program logic for single-assignment programs. Unlike Hoare logic (including the goal-directed system Hg), this single-assignment logic admits a single derivation for each valid Hoare triple, by imposing a strategy based on *forward propagation* of assertions. The system takes advantage of the characteristics of SA programs not only by dispensing with variable substitution,

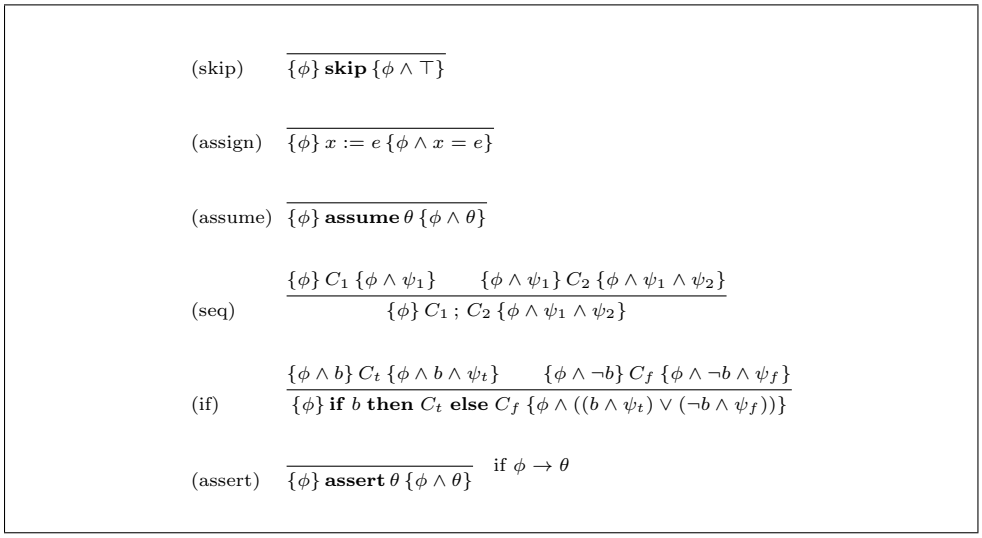

Fig. 5. Forward version of Hoare logic for SA programs – System $\mathsf{H_{sa}}$

but also by producing derivations that are free of the problem of exponential explosion of the size of formulas. As such, it can be used to systematically generate verification conditions for SA programs, and it will in fact allow us to prove the soundness and completeness of the $\mathsf{VC^{sp}}$ function of Definition 6 from Section 2.

Figure 5 contains the rules of system $\mathsf{H_{sa}}$. Like system Hg it is goal-directed since it does not contain a consequence rule. We first note the following:

LEMMA 35. *Let* $C \in \mathbf{Comm}^{\mathrm{SA}}$ *and* ϕ, ψ, $\psi' \in \mathbf{Assert}$ *such that* $\phi \# C$, $\vdash_{\mathsf{H_{sa}}} \{\phi\}\,C\,\{\psi\}$, *and* $\vdash_{\mathsf{H_{sa}}} \{\phi\}\,C\,\{\psi'\}$. *Then*

(1) $\psi' = \psi$;

(2) $\mathsf{FV}(\psi) \subseteq \mathsf{FV}(\phi) \cup \mathsf{Vars}(C)$.

PROOF. Straightforward induction on the structure of the derivation of $\vdash_{\mathsf{H_{sa}}} \{\phi\}\,C\,\{\psi\}$. □

We prove soundness of this system directly with respect to system H:

PROPOSITION 36 SOUNDNESS OF $\mathsf{H_{sa}}$. *Let* $C \in \mathbf{Comm}^{\mathrm{SA}}$ *and* $\phi, \psi \in \mathbf{Assert}$ *such that* $\phi \# C$ *and* $\vdash_{\mathsf{H_{sa}}} \{\phi\}\,C\,\{\psi\}$. *Then* $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi\}$.

PROOF. By induction on the structure of the derivation. For the inductive cases we note that the assigned variables condition of the induction hypotheses follow from $\phi \# C$, using Lemma 35 in the case of the second hypothesis of (seq): in $\vdash_{\mathsf{H_{sa}}} \{\phi\}\,C_1\,\{\phi \wedge \psi_1\}$ we have from the lemma that $\mathsf{FV}(\phi \wedge \psi_1) \subseteq \mathsf{FV}(\phi) \cup \mathsf{Vars}(C_1)$, and thus in $\vdash_{\mathsf{H_{sa}}} \{\phi \wedge \psi_1\}\,C_2\,\{\phi \wedge \psi_1 \wedge \psi_2\}$ we have $(\phi \wedge \psi_1) \# C_2$, since $C_1\,;\,C_2 \in \mathbf{Comm}^{\mathrm{SA}}$ (thus variables occurring in C_1 are not assigned in C_2), and moreover $\phi \# (C_1\,;\,C_2)$. □

The completeness of $\mathsf{H_{sa}}$ on the other hand will be established with respect to the

goal-directed system Hg. Observe that the $\mathsf{H_{sa}}$ system is not capable of deriving every valid triple, and the completeness result takes this into account.

PROPOSITION 37 COMPLETENESS OF $\mathsf{H_{sa}}$. *Let* $C \in \mathbf{Comm}^{\text{SA}}$ *and* $\phi, \psi \in \mathbf{Assert}$ *such that* $\phi \# C$ *and* $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$. *Then* $\vdash_{\mathsf{H_{sa}}} \{\phi\}\, C\, \{\phi \wedge \psi'\}$ *for some* $\psi' \in \mathbf{Assert}$ *such that* $\models \phi \wedge \psi' \rightarrow \psi$.

PROOF. The proof proceeds by induction on the structure of C. We show here two cases. As an example of a base case, consider $\vdash_{\mathsf{Hg}} \{\phi\}\, x := e\, \{\psi\}$. This happens exactly when $\models \phi \rightarrow \psi[e/x]$. Now system $\mathsf{H_{sa}}$ derives $\{\phi\}\, x := e\, \{\phi \wedge x = e\}$, so it suffices to note that $\phi \rightarrow \psi[e/x]$ entails $\phi \wedge x = e \rightarrow \psi$, since $x \notin \mathsf{FV}(e)$.

Consider now the inductive case when $\vdash_{\mathsf{Hg}} \{\phi\}\, C_1\, ;\, C_2\, \{\psi\}$. Then it must be, for some intermediate assertion θ, $\vdash_{\mathsf{Hg}} \{\phi\}\, C_1\, \{\theta\}$ and $\vdash_{\mathsf{Hg}} \{\theta\}\, C_2\, \{\psi\}$. Note that since $\phi \#(C_1\, ;\, C_2)$ then also $\phi \# C_1$, and the following induction hypothesis applies for C_1:

IH1. If $\phi \# C_1$ and $\vdash_{\mathsf{Hg}} \{\phi\}\, C_1\, \{\theta\}$ then $\vdash_{\mathsf{H_{sa}}} \{\phi\}\, C_1\, \{\phi \wedge \theta'\}$ for some formula θ' such that $\models \phi \wedge \theta' \rightarrow \theta$.

Applying Lemma 25 we have from $\vdash_{\mathsf{Hg}} \{\theta\}\, C_2\, \{\psi\}$ and $\models \phi \wedge \theta' \rightarrow \theta$ that $\vdash_{\mathsf{Hg}} \{\phi \wedge \theta'\}\, C_2\, \{\psi\}$. Note also that by Lemma 35 we have that $\mathsf{FV}(\phi \wedge \theta') \subseteq \mathsf{FV}(\phi) \cup \mathsf{Vars}(C_1)$, and thus $(\phi \wedge \theta') \# C_2$. We apply the following induction hypothesis regarding C_2:

IH2. If $(\phi \wedge \theta') \# C_2$ and $\vdash_{\mathsf{H}} \{\phi \wedge \theta'\}\, C_2\, \{\psi\}$ then $\vdash_{\mathsf{H_{sa}}} \{\phi \wedge \theta'\}\, C_2\, \{\phi \wedge \theta' \wedge \psi'\}$ for some formula ψ' such that $\models \phi \wedge \theta' \wedge \psi' \rightarrow \psi$.

Thus applying the (seq) rule of $\mathsf{H_{sa}}$ we have $\vdash_{\mathsf{H_{sa}}} \{\phi\}\, C_1\, ;\, C_2\, \{\phi \wedge \theta' \wedge \psi'\}$, with $\models \phi \wedge \theta' \wedge \psi' \rightarrow \psi$. □

The significance of system $\mathsf{H_{sa}}$ is twofold: first, it offers an algorithm to check if a given triple is valid. To check the validity of $\{\phi\}\, C\, \{\psi\}$ (where $\phi \# C$) one attempts to construct a derivation with root $\{\phi\}\, C\, \{\phi \wedge \psi'\}$ for some formula ψ'. This may not be possible if there exists an execution in which an assert statement fails (in which case the side condition of an assert rule is not valid), but if all side conditions are valid, then the derivation is unique (and ψ' is uniquely generated). There is now one more thing to do, which is to check the validity of $\phi \wedge \psi' \rightarrow \psi$. If this formula is valid then so is the Hoare triple.

The second significant fact is that, if it exists, this unique derivation avoids the exponential explosion of the size of the formulas manipulated: observe that in the (if) rule, the postcondition of the triple in the conclusion does not contain two copies of the precondition ϕ, but a single one.

4.3 Verification Conditions for SA Programs

The *verification conditions* generated for a triple in system $\mathsf{H_{sa}}$ come from the side conditions corresponding to the assert statements in the program, together with a formula of the form $\phi \wedge \psi' \rightarrow \psi$: the triple is derivable in system $\mathsf{H_{sa}}$ if and only if all of these conditions are valid. As shown in Section 3.5, for standard imperative programs it is possible to bypass completely the construction of any derivation, by writing a VCGen algorithm that simply collects these conditions.

$$
\begin{aligned}
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{skip}) &= (\top, \emptyset) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, x := e) &= (x = e, \emptyset) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{assume}\,\theta) &= (\theta, \emptyset) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, C_1\,;\, C_2) &= (F_1 \wedge F_2, V_1 \cup V_2) \\
&\quad \text{where} \\
&\quad (F_1, V_1) = \mathsf{VC}^{\mathsf{sp}}(\phi, C_1) \\
&\quad (F_2, V_2) = \mathsf{VC}^{\mathsf{sp}}(\phi \wedge F_1, C_2) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{if}\ b\ \mathbf{then}\ C^t\ \mathbf{else}\ C^f) &= ((b \wedge F^t) \vee (\neg b \wedge F^f), V^t \cup V^f) \\
&\quad \text{where} \\
&\quad (F^t, V^t) = \mathsf{VC}^{\mathsf{sp}}(\phi \wedge b, C^t) \\
&\quad (F^f, V^f) = \mathsf{VC}^{\mathsf{sp}}(\phi \wedge \neg b, C^f) \\
\mathsf{VC}^{\mathsf{sp}}(\phi, \mathbf{assert}\,\theta) &= (\theta, \{\phi \to \theta\})
\end{aligned}
$$

Fig. 6. Verification conditions generator for SA programs

The VCGen for system $\mathsf{H_{sa}}$ is the function $\mathsf{VC}^{\mathsf{sp}}$ of Definition 8, which we reproduce for convenience in Figure 6. Observe that our previous remark on how exponential explosion of the size of the formulas is avoided in $\mathsf{H_{sa}}$ derivations is closely linked to the efficiency of $\mathsf{VC}^{\mathsf{sp}}$, discussed at length in Section 2. It is straightforward to prove the following correspondence between this VCGen and system $\mathsf{H_{sa}}$.

Proposition 38 Soundness and Completeness of $\mathsf{VC}^{\mathsf{sp}}$. *Let* $C \in \mathbf{Comm}^{\mathrm{SA}}$, $\phi, \psi, F \in \mathbf{Assert}$ *and* $V \subseteq \mathbf{Assert}$, *such that* $(F, V) = \mathsf{VC}^{\mathsf{sp}}(\phi, C)$. *Then:*

(1) *If* $\models V$ *then* $\vdash_{\mathsf{H_{sa}}} \{\phi\}\, C\, \{\phi \wedge F\}$
(2) *If* $\vdash_{\mathsf{H_{sa}}} \{\phi\}\, C\, \{\phi \wedge \psi'\}$ *then* $\models V$ *and* $\psi' \equiv F$

Proof.

(1) By induction on the structure of the program C. We show the inductive case when C is $C_1\,;\, C_2$. Since we have $\phi \# (C_1\,;\, C_2)$ then also $\phi \# C_1$, and the following induction hypothesis applies for C_1:
IH1. If $\models V_1$ then $\vdash_{\mathsf{H_{sa}}} \{\phi\}\, C_1\, \{\phi \wedge F_1\}$, with $(F, V) = \mathsf{VC}^{\mathsf{sp}}(\phi, C_1)$.
Now by Lemma 35 we have that $\mathsf{FV}(\phi \wedge F_1) \subseteq \mathsf{FV}(\phi) \cup \mathsf{Vars}(C_1)$, and thus $(\phi \wedge F_1) \# C_2$. We apply the following induction hypothesis regarding C_2:
IH2. If $\models V_2$ then $\vdash_{\mathsf{H_{sa}}} \{\phi \wedge F_1\}\, C_2\, \{\phi \wedge F_1 \wedge F_2\}$, with $(F_2, V_2) = \mathsf{VC}^{\mathsf{sp}}(\phi \wedge F_1, C_2)$.
And applying the (seq) rules yields $\vdash_{\mathsf{H_{sa}}} \{\phi\}\, C_1\,;\, C_2\, \{\phi \wedge F_1 \wedge F_2\}$.
(2) Straightforward induction on the structure of the derivation.

□

Combined, the previous results imply that the mechanism introduced in Section 2.4 for generating verification conditions for single-assignment programs with assert and assume statements is both sound and complete:

PROPOSITION 39. *Let* $C \in \mathbf{Comm}^{\mathrm{SA}}$, $\phi, \psi, F \in \mathbf{Assert}$ *and* $V \subseteq \mathbf{Assert}$, *such that* $\phi \# C$ *and* $(F, V) = \mathsf{VC}^{\mathsf{sp}}(\phi, C)$. *Then*

$$\models \{\phi\}\, C\, \{\psi\} \qquad \textit{iff} \qquad \models V,\ \phi \wedge F \rightarrow \psi$$

PROOF.

If part.. By Proposition 38 (1), we have $\vdash_{\mathsf{H_{sa}}} \{\phi\}\, C\, \{\phi \wedge F\}$, thus by Proposition 36 $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\phi \wedge F\}$. But from $\models \phi \wedge F \rightarrow \psi$ we have using the consequence rule $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$, and then $\models \{\phi\}\, C\, \{\psi\}$ by the soundness of system H.

Only if part.. $\models \{\phi\}\, C\, \{\psi\}$ implies by completeness of system H that $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$, and (by Proposition 26) $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$. Then (Proposition 37) $\vdash_{\mathsf{H_{sa}}} \{\phi\}\, C\, \{\phi \wedge \psi'\}$ for some ψ' such that $\models \phi \wedge \psi' \rightarrow \psi$, and by Proposition 38 (2) $\models V$ and $\models \phi \wedge F \rightarrow \psi$.

□

4.4 Program Verification Using Intermediate SA Form

We will now put up a framework for the verification of branching programs, based on their translation to intermediate single-assignment form and the subsequent generation of efficient verification conditions from this intermediate code. The resulting verification technique is sound and complete.

We could at this point give one particular algorithm for translating standard programs into single-assignment form, but for the sake of generality we prefer to characterize the properties that such a translation should have. A translation of branching programs into an intermediate SA form must of course abide by the syntactic restrictions on single-assignment annotated programs, but it must also obey additional requirements of a semantic nature: the translation must be sound (it will not translate invalid triples into valid SA triples) and complete (it translates valid triples into valid SA triples).

DEFINITION 40 SA TRANSLATION.
A function $\mathcal{T} : \mathbf{Assert} \times \mathbf{Comm} \times \mathbf{Assert} \rightarrow \mathbf{Assert} \times \mathbf{Comm}^{\mathrm{SA}} \times \mathbf{Assert}$ *is a* single-assignment translation of Hoare triples *if when* $(\phi', C', \psi') = \mathcal{T}(\phi, C, \psi)$ *we have* $\phi' \# C'$, *and the following holds:*

$$\models \{\phi'\}\, C'\, \{\psi'\} \quad \textit{iff} \quad \models \{\phi\}\, C\, \{\psi\}$$

EXAMPLE 41. *Consider the Hoare triple*

```
{y == yaux and x = K}
   if (x>0) then y := y+1 else y := 0 end;
   assert y == yaux+1;
   if (x>0) then y := y+1 else y := 0 end;
   assert y == yaux+2;
   if (x>0) then y := y+1 else y := 0 end;
{y == yaux+3}
```

The following is a possible translation.

```
{y0 == yaux and x0 == K}
   if (x0>0) then y1 := y0+1 else y1 := 0 end;
   assert y1 == yaux+1;
```

```
    if (x0>0) then y2 := y1+1 else y2 := 0 end;
    assert y2 == yaux+2;
    if (x0>0) then y3 := y2+1 else y3 := 0 end;
{y3 == yaux+3}
```

It is clearly in accordance with the previous definition: the resulting program is in SA form; it does not assign variables occurring free in the precondition, and it is valid if and only if the initial triple is valid (depending only on the value of K).

Operationally, the obvious way to implement an SA translation is to use indexed families of variables and traverse the program keeping a table of current indexes of the variables, incrementing them when new assignments occur. But note that this is merely a design choice, not in any way required by the definitions. The bottom line is that the translation must be shown to comply with Definition 40. A detailed example of such a translation, together with the proof that it complies to the requirements of the definition, can be found in [Lourenço et al. 2015b].[6]

When combined with an SA translation of programs, the VCGen of the previous section yields a mechanism for the verification of standard (i.e. non-SA) programs.

PROPOSITION 42. *Let* $C \in \mathbf{Comm}$, $C' \in \mathbf{Comm}^{\mathrm{SA}}$, $\phi, \phi', \psi, \psi', F \in \mathbf{Assert}$ *and* $V \subseteq \mathbf{Assert}$, *such that* $\{\phi'\}\, C'\, \{\psi'\} = \mathcal{T}(\phi, C, \psi)$, *and* $(F, V) = \mathsf{VC}^{\mathsf{sp}}(\phi', C')$. *Then*

$$\models \{\phi\}\, C\, \{\psi\} \qquad \textit{iff} \qquad \models V,\ \phi' \wedge F \rightarrow \psi'$$

PROOF. Follows directly from Proposition 39 and Definition 40. □

Note that in Section 2 no preconditions or postconditions were used (they could of course always be introduced as an initial assume and a final assert statements). In this case correctness corresponds to the validity of a triple $\{\top\}\, C\, \{\top\}$, and the above result can be simplified to

$$\models \{\top\}\, C\, \{\top\} \qquad \text{iff} \qquad \models V$$

5. CONCLUSION AND FURTHER TOPICS

We have formalized the verification technique that consists in translating code to single-assignment form and then generating verification conditions from this intermediate form. We have established the expected soundness and completeness properties for this technique, and studied in detail different methods for generating VCs from SA code not containing iteration, as well as relations between them.

We end our study with an explanation of how the technique can be used to verify programs containing iteration, and showing that the use of intermediate SA forms makes sense (and has additional advantages) in the context of the richer specification languages that are typical of deductive verification tools. Finally, we explain another major advantage of using intermediate SA form for program verification, which is the fact that the program logic for single-assignment programs enjoys the *adaptation-completeness* property.

[6] In fact this translation applies to programs containing iteration (see Section 5.1), so its requirements are somewhat different, but closely related.

5.1 Iteration: SA Loops and Bounded Model Checking

Admittedly, the simplicity of the programming language considered in this paper stands very distant from the sophistication of modern programming languages. Still, the study of VC generation for a simplistic fragment is extremely important, on one hand because the potential explosion of the size of the VCs is caused by the control flow, not by complex programming language features, and on the other hand because it is well-known that many features, in particular a complex memory model, can be encoded logically and incorporated in mechanisms like the ones studied in the previous sections (their treatment is basically orthogonal). See for instance [Kroening 2009] for details on the logical encoding of C pointers and arrays. Having said this, one feature that was absent in sections 2 and 4 is so fundamental that, without considering it, we cannot claim to have covered the basics of VC generation for imperative programs. This feature is of course *iteration*.

Loops are covered by Hoare logic (see Section 3), and in fact the entire verification workflow of Section 4, based on intermediate single-assignment form, can be extended to programs with loops – we do this in detail in [Lourenço et al. 2015a] by introducing a notion of SA loop. Observe that, strictly speaking, iteration is incompatible with the single-assignment principle, because of the need to communicate values between subsequent iterations. Take for instance the following loop that calculates exponentiation:

```
r := 1;
while (n>0) {
  r := r*x;
  n := n-1
}
```

A naive attempt to convert it to SA form would simply translate the body of the loop:

```
r1 := 1;
while (n0>0) {
  r2 := r1*x0;
  n1 := n0-1
}
```

This is clearly wrong since `r2` and `n1` are assigned the same values in every iteration, and the value of the variable `n0` occurring in the loop condition is never modified. The solution is to allow for variables to be assigned more than once, but in a controlled way. We use `for` loops for this, and allow the 'update' field of loop headers to assign variables that occur in the loop body. The above example could be translated as follows:

```
r1 := 1;
for (n1:=n0; r2:=r1, n1>0, n1:=n2; r2:=r3) {
  r3 := r2*x0;
  n2 := n1-1
}
```

Note that the 'initialization' field in the loop header is used to introduce fresh version variables; these same versions will be updated between iterations, and used in the loop condition. They will also be used in the loop invariants required for deductive reasoning.

As shown in [Lourenço et al. 2015a], it is possible to define a logic for SA programs containing loops of this form, and to extend the verification workflow of Section 4 based on this logic. A translation of programs and Hoare triples into SA form must now work on *annotated programs*, translating loop invariants in addition to the code. The requirements on such a translation (Definition 40) must also be modified: to achieve completeness of the approach, annotated programs (and in particular loop invariants) must be translated in a way that preserves derivability of Hoare triples.

Bounded Model Checking of Software. An alternative to deductive verification is the exploration of a bounded-size model of the code. This verification technique [Clarke et al. 2004] differs from deductive verification in that it is fully automated, not requiring users to provide loop invariants or other program annotations. Instead, it is based on *loop unwinding*: given a bound K, a bounded model checking tool will unwind K iterations of every loop in the program, thus obtaining a branching program from which verification conditions can be generated, applying any of the methods described in Section 2.

Described in this way, bounded model checking in inherently unsound: the violation of a property deep within the execution of a loop may not be detected if the loop has not been sufficiently expanded before the logical model is generated. To regain soundness, an *unwinding assertion* with the negated loop condition is introduced at the end of each expanded loop. If all these assertions are successfully passed, this means that the bound K was sufficient to encompass all possible executions of the loop. The technique becomes sound, but loses completeness: it may now be the case that an unwinding assertion is violated to signal the existence of executions that are not covered by the model within the bound K, when in fact none of the user assertions in the program are violated in any execution.

Bounded model checking of software reinforces the idea that it may well be worth comparing empirically the different VCGens of Section 2. Loop unwinding easily generates formulas of a very considerable size, and it is precisely in the presence of such formulas that the choice of logical encoding may affect the performance of satisfiability solvers.

5.2 SA Form and Behavioral Interface Specification Languages

In this paper we have considered a specification mechanism consisting only in the use of assume and assert commands in the code. This is most commonly used with software model checking tools. Deductive verification tools usually employ richer specification languages, known as *Behavioral Interface Specification Languages* (BISLs) [Hatcliff et al. 2012].

In these languages a *contract* consisting of a precondition and a postcondition is usually associated with each program unit (function, procedure, or method). Contracts may possibly contain other elements, such as a *frame condition* specifying the effect of that program unit on the global state. We will now see how some

features of these languages are particularly easy to handle if the code is translated into an intermediate single-assignment form.

Consider for instance a (parameterless) procedure **exp** that computes exponentiation. The input variables are x and n, and the output is written in a variable r. The specification (or *contract*) of such a procedure might be:

```
@requires n >= 0
@ensures r = x^n
exp {
  r := 1;
  while (n > 0) {
    r := r*x;
    n := n-1
  }
}
```

corresponding to the Hoare triple

$$\{n \geq 0\}\, \mathbf{call}\, \mathbf{exp}\, \{r = x^n\} \tag{4}$$

In fact this specification is not adequate, since the values of x and n may be modified by the procedure (this is indeed the case for n in the above implementation), and the postcondition should refer to the *initial values* of these variables. It suffices to think that the above specification would have as trivial solution the program that simply assigns 0 to n and 1 to r.

In Hoare logic one may use *auxiliary variables* (occurring only in assertions, not used by the program) to refer to the value of a variable in the pre-state of a program or procedure. We will employ capital letters for auxiliary variables:

$$\{n \geq 0 \wedge x = X \wedge n = N\}\, \mathbf{call}\, \mathbf{exp}\, \{r = X^N\} \tag{5}$$

Common behavioral interface specification languages like JML, ACSL, or the SPARK specification language offer an alternative to auxiliary variables: a specific notation that can be used to fetch the value of a variable (or expression in general) in the pre-state. For instance,

```
@requires n >= 0
@ensures r = \old(x^n)
exp { ... }
```

But consider now that this code is translated into an intermediate single-assignment form (transparently to the user). In this form the procedure's contract might look like the following:

```
@requires n0 >= 0
@ensures rf = x0^n0
exp { ... }
```

where the indexes 0, f are used respectively for the initial and final version variables. Note how easy it is to translate the specification into the SA form: in the pre and postcondition each variable is replaced by its relevant version (initial and

final respectively); the `\old` operator used in the postcondition simply forces each variable in its argument expression to be translated to its initial version.

Specification languages usually offer a generalized form of the `\old` operator introduced above, allowing to fetch the value of a variable not only in the pre-state but in arbitrary program states. Consider the fragment

```
@requires x > 0
@ensures x - \at(x, L1) > 10
proc1 {
    x := x + 10;
L1: x := x + 20;
    x := x + 30;
    @assert x>0;
    x := x + 40
}
```

This illustrates the ability to refer to the value of a variable at a given point of the program (identified through a label, `L1` in the example), using the `\at` operator. The code also illustrates the insertion of arbitrary assertions to be checked anywhere in the code (as used extensively throughout this paper), which is also present in BISLs. The translation of this annotated code into single-assignment form could look like:

```
@requires x0 > 0
@ensures x4 - x2 > 10
proc1 {
    x1 := x0 + 10;
L1: x2 := x1 + 20;
    x3 := x2 + 30;
    @assert x3>0;
    x4 := x3 + 40
}
```

This immediately highlights how easily the first feature is implemented by tools based on intermediate SA code: the notation `\at(x,L1)` simply forces the variable to be translated into the appropriate version at that point of the program.

5.3 SA Form and the Adaptation Problem

The use of auxiliary variables does not mix well with the consequence rule. Say you now want to derive from the updated specification of our example procedure the following Hoare triple

$$\{x = z \wedge n = 10\}\, \mathbf{call}\, \mathbf{exp}\, \{r = z^{10}\} \tag{6}$$

This should be obvious, but it does not follow from (5) by the consequence rule, since the conditions

$$x = z \wedge n = 10 \;\rightarrow\; n \geq 0 \wedge x = X \wedge n = N \tag{7}$$

$$r = X^N \;\rightarrow\; r = z^{10} \tag{8}$$

are not valid. In both conditions the auxiliary variables X and N are of course free, and they do not perform the intended job of transporting information from the pre-state to the post-state.

A program logic is said to be *adaptation-complete* if whenever the validity of a triple $\{\phi\}\, C\, \{\psi\}$ entails the validity of another triple $\{\phi'\}\, C\, \{\psi'\}$ (assuming the specification (ϕ, ψ) is satisfiable), then the latter triple is derivable from the former in the deductive system of the logic. The above example shows that Hoare logic does not enjoy this property, since the validity of the triple (5) indeed implies the validity of (6).

Now suppose the procedure is translated to SA form, with the contract:

```
@requires n0 >= 0
@ensures rf = x0^n0
exp { ... }
```

The Hoare triple (6) involving a call to the procedure would be translated as:

$$\{x_0 = z \wedge n_0 = 10\}\, \mathbf{call\ exp}\, \{r_f = z^{10}\} \tag{9}$$

The consequence rule would now produce the VCs

$$x_0 = z \wedge n_0 = 10 \;\rightarrow\; n_0 \geq 0 \tag{10}$$

$$r_f = x_0^{n_0} \;\rightarrow\; r_f = z^{10} \tag{11}$$

The first VC is certainly valid (since the right-hand side does not contain auxiliary variables), but not the second: information from the pre-state is missing.

But the remarkable fact is that, if the Hoare triple $\{\phi\}\, C\, \{\psi\}$ results from the translation of some triple into single-assignment form, then C does not assign variables occurring free in ϕ, and in this case *weaker versions of the consequence rule can be used*. In particular, the postcondition can be weakened as follows:

$$\frac{\{\phi\}\, C\, \{\psi\}}{\{\phi\}\, C\, \{\psi'\}} \text{ if } \; \phi \wedge \psi \rightarrow \psi'$$

This follows from the *monotonicity* of the logic: if the program C does not assign free variables of the precondition ϕ, its execution can never modify the truth value of ϕ, which remains true in the post-state.

So starting from the procedure's contract we could have, applying the standard consequence rule with side condition 10:

$$\{x_0 = z \wedge n_0 = 10\}\, \mathbf{call\ exp}\, \{r_f = x_0^{n_0}\} \tag{12}$$

and then using the above rule:

$$\{x_0 = z \wedge n_0 = 10\}\, \mathbf{call\ exp}\, \{r_f = z^{10}\} \tag{13}$$

since the condition $x_0 = z \wedge n_0 = 10 \wedge r_f = x_0^{n_0} \;\rightarrow\; r_f = z^{10}$ is valid.

In reality the example would have to be treated in a more complicated way, since each single-assignment procedure will use its own (disjoint) set of versions for each global variable in the initial translated program. But the bottom line is that this little example illustrates how the adaptation problem of Hoare logic can be naturally solved in the single-assignment setting. The logic for SA programs

introduced in [Lourenço et al. 2015a], which extends the logic of Section 4 with iteration, is carefully designed to be monotonic in the above sense, and indeed we prove that it enjoys adaptation-completeness.

The monotonicity of the program logic for single-assignment programs has other advantages. In Hoare logic loop invariants often have to include information that is not relevant for the computations performed in the loop (for instance regarding data that is not read or written in the loop body). There is no way to transport this information from the pre-state to the post-state of the loop without explicitly including it in the loop invariant – such properties are sometimes called *continuous invariants*. In the SA setting this is not required: properties corresponding to formulas whose free variables are not assigned in the loop body are transported automatically to the post-state.

REFERENCES

Almeida, J. B., Frade, M. J., Pinto, J. S., and de Sousa, S. M. 2011. *Rigorous Software Develpment*. Springer-Verlag London Ltd. First edition.

Apt, K. R. 1981. Ten years of Hoare's logic: A survey - part 1. *ACM Trans. Program. Lang. Syst. 3,* 4, 431–483.

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds. Lecture Notes in Computer Science, vol. 4111. Springer, 364–387.

Clarke, E., Kroening, D., and Lerda, F. 2004. A tool for checking ANSI-C programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 168–176.

Cook, S. A. 1978. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput. 7,* 1, 70–90.

Cytron, R., Ferrante, J., Rosen, B., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems 13,* 4 (Oct.), 451–490.

Dijkstra, E. W. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey.

Flanagan, C. and Saxe, J. B. 2001. Avoiding exponential explosion: generating compact verification conditions. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 193–205.

Frade, M. J. and Pinto, J. S. 2011. Verification Conditions for Source-level Imperative Programs. *Computer Science Review 5*, 252–277.

Hatcliff, J., Leavens, G. T., Leino, K. R. M., Müller, P., and Parkinson, M. 2012. Behavioral interface specification languages. *ACM Comput. Surv. 44,* 3 (June), 16:1–16:58.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12*, 576–580.

Kroening, D. 2009. Software Verification. In *Handbook of Satisfiability*. 505–532.

Leino, K. R. M. 2005. Efficient weakest preconditions. *Inf. Process. Lett. 93,* 6, 281–288.

Loeckx, J. and Sieber, K. 1987. *The foundations of program verification (2nd ed.)*. John Wiley & Sons, Inc., New York, NY, USA.

Lourenço, C. B., Frade, M. J., and Pinto, J. S. 2015a. Single-assignment program verification. Submitted for publication.

Lourenço, C. B., Frade, M. J., and Pinto, J. S. 2015b. A single-assignment translation for while annotated programs. Unpublished manuscript available from `http://www.di.uminho.pt/~mjf/pub/SAtranslation.pdf`.

Vanbroekhoven, P., Janssens, G., Bruynooghe, M., and Catthoor, F. 2007. A practical dynamic single assignment transformation. *ACM Trans. Des. Autom. Electron. Syst. 12*.