Formalizing Single-assignment Program Verification: an Adaptation-complete Approach

Cláudio Belo Lourenço, Maria João Frade, and Jorge Sousa Pinto

HASLab/INESC TEC & Universidade do Minho, Portugal

Abstract. Deductive verification tools typically rely on the conversion of code to a single-assignment (SA) form. In this paper we formalize program verification based on the translation of *While* programs annotated with loop invariants into a dynamic single-assignment language with a dedicated iterating construct, and the subsequent generation of compact, indeed linear-size, verification conditions. Soundness and completeness proofs are given for the entire workflow, including the translation of annotated programs to SA form. The formalization is based on a program logic that we show to be *adaptation-complete*. Although this important property has not, as far as we know, been established for any existing program verification tool, we believe that adaptation-completeness is one of the major motivations for the use of SA form as an intermediate language. Our results here show that indeed this allows for the tools to achieve the maximum degree of adaptation when handling subprograms.

1 Introduction

In the last years deductive program verification has reached a stage of a certain maturity, to the point that a number of tools are now available allowing users to prove properties of programs written in real-world languages like C, Java, or SPARK [3,6,10,25]. Deductive techniques attempt to establish the correctness of a software system with respect to a specification, usually given as a set of *contracts* expressed in first-order logic. Their precision depends on information provided by the user in the form of *annotations*, in particular *loop invariants*.

Three trends have characterized the development of modern program verifiers: first, they employ Satisfiability Modulo Theories (SMT) solvers to check the validity of first-order formulas. The nuclear component is a Verification Conditions Generator (VCGen), that takes as input a program and a specification, and produces a set of first-order proof obligations that are sent to a solver. If all the conditions are valid, then the program is correct. The second trend is that deductive verification tools are usually *generic*, based on programming languages tailored for verification. Rather than producing from scratch a dedicated verifier, programs of a particular language are translated into the intermediate language of the tool, together with a background encoding of the relevant aspects of that language. Two widely used generic verifiers are Boogie [4] and Why3 [13].

Finally, and this is the subject of the present paper, modern tools employ internally a *Single-Assignment* (SA) representation of the code [11], in which variables may not be assigned after they have been read or written. Not only are SA branching programs easier to encode logically, but its use also solves a fundamental inefficiency issue. Verification conditions generated from standard imperative code may be of *exponential size* in the size of the programs, which destroys any hope of effectively verifying reasonably-sized programs. However, Flanagan and Saxe [14] have shown that conversion of the code to SA form allows for the generation of quadratic size VCs (a technique that achieves conditions of linear-size with respect to the SA program was later proposed [5]). Other advantages have to do with the fact that intermediate values of computations are never lost: when an instruction like $x := x + 1$ is executed, one variable will store the initial value of x, and a new variable will store its new value. Specification languages like ACSL, for ANSI-C [7], often allow the value of a variable at a given program point to be used in assertions. In an SA setting this amounts to simply fetching the adequate "version variable". This also means that *continuous invariants* (that are not relevant for a loop but may be required after it) are transported automatically, and do not have to be explicitly included.

On the theoretical side, the foundations of program verification have traditionally lied in two different frameworks: Dijkstra's predicate transformers for a *guarded commands* language [12] and program logics, like Hoare logic [17] and separation logic [26]. Guarded commands have been used as an intermediate language in tools like ESC/Java [22] and more recently the Boogie generic verifier. Many pragmatic aspects of program verification have been addressed and described in this setting, in particular the generation of efficiently provable Verification Conditions (VCs) and the treatment of unstructured programs [14,5]. The program logic tradition on the other hand, which is based on separate operational and axiomatic semantics of programming languages, has allowed for the study of properties like soundness and (relative) completeness of Hoare logic with respect to the standard semantics of a While language [2], an approach that has been extended with the treatment of pointers and aliasing in separation logic.

An important issue is that of *modular verification* and proof reuse. Ideally, one produces a separate proof of correctness for each occurrence (or call) of a subprogram C inside a program P, and then *adapts* the proved specification of C to different 'local' specifications. A formalism that always allows for this to be done is said to be *adaptation-complete* [20]; in its original formulation Hoare logic is not adaptation-complete. This is a problem in the presence of recursive procedures, since it leads to incompleteness of the program logic itself, but it is also a problem for the implementation of tools where the correctness of each procedure is proved once and for all with respect to a *contract* that must be adapted to the local context of each call to it. Our work shows that adaptation-completeness is a natural property of reasoning in the single-assignment setting.

Contributions. In this paper we formalize a verification technique for *While* programs annotated with invariants, based on their conversion to an intermediate SA form. Verification tools convert programs to single-assignment form internally and profit from this in various ways, in particular to achieve efficiency and to handle subprograms – our technique is a minimal model of such a tool. It relies on (i) a novel notion of single-assignment program that supports loops annotated with invariants; (ii) a notion of translation of *While* programs annotated with loop invariants (resp. Hoare triples containing such programs) into SA programs (resp. Hoare triples containing SA programs); (iii) a Hoare-style logic for these programs; and (iv) a VCGen generating *linear-size* verification conditions for Hoare triples containing SA programs. The entire workflow is proved to be sound and complete – in particular, we show how invariants annotated in the initial *While* program are translated into the intermediate SA form in a way that guarantees the completeness of the approach. This means that if the invariants annotated in the original program are appropriate for showing its correctness, then the verification of the translated SA program will be successful.

An adaptation-complete variant of the logic is also proposed, by adding to the inference system a dedicated consequence rule with a simple side condition. This new consequence rule is restricted to reasoning about triples in which the program does not assign any variable occurring free in the precondition; since the Hoare logic for SA programs propagates preconditions forward in a way that preserves this property, the rule can be applied at any point in a derivation. It provides the highest degree of adaptation, without the need to check any additional complicated conditions or rules, as used to be the case in adaptation-complete presentations of Hoare logic [2,1,20].

As an added bonus, this paper can also be seen as bridging a gap between two different theoretical traditions – the guarded commands/predicate transformers setting, where the use of single-assignment form was first introduced for the sake of proof efficiency, and the Hoare logic tradition, that formalizes reasoning with loop invariants based on a standard interpretation of imperative programs.

The paper is organized as follows: Section 2 contains background material. In Section 3 we introduce a language of iterating SA programs: loops are annotated with invariants; they have single-assignment bodies; and a *renaming* allows for the values of the initial variables to be updated between iterations. We propose a Hoare-style partial correctness program logic for this language in Section 4; its inference system admits only derivations guided by the annotated loop invariants, following a forward-propagation strategy. We also give an algorithm that generates compact conditions (in the sense of Flanagan and Saxe [14]) for a given Hoare triple, and then optimize it to generate *linear-sized* VCs. The next sections contain our main results. We first consider the verification workflow based on the translation of annotated While programs to the SA language. We identify, in Section 5, the semantic requirements that are expected from such a translation. The workflow is validated by showing that the generation of VCs from the SA form is sound and complete for the verification of the initial program (a concrete translation is given in the appendix, together with the proof that it meets the requirements). In Section 6 we show how the program logic can be extended with a special consequence rule that makes it *adaptation-complete.* Finally Section 7 discusses related work and Section 8 concludes the paper.

2 Hoare Logic

We briefly review Hoare logic for While programs. The logic deals with the notion of correctness of a program w.r.t. a *specification*.

Syntax. We consider a typical While language whose commands $C \in \mathbf{Comm}$ are defined over a set of variables $x \in \mathbf{Var}$ in the following way:

$$C \ ::= \ \mathbf{skip} \ \mid \ x := e \ \mid \ C \, ; C \mid \ \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \ \mid \ \mathbf{while}\ b\ \mathbf{do}\ C$$

We will not fix the language of program expressions $e \in \mathbf{Exp}$ and Boolean expressions $b \in \mathbf{Exp^{bool}}$, both constructed over variables from $\mathbf{Var}$ (a standard instantiation is for $\mathbf{Exp}$ to be a language of integer expressions and $\mathbf{Exp^{bool}}$ constructed from comparison operators over $\mathbf{Exp}$, together with Boolean operators). In addition to expressions and commands, we need formulas that express properties of particular states of the program. Program assertions $\phi, \theta, \psi \in \mathbf{Assert}$ (preconditions and postconditions in particular) are formulas of a first-order language obtained as an expansion of $\mathbf{Exp^{bool}}$.

We also require a class of formulas for specifying the behaviour of programs. Specifications are pairs (ϕ, ψ), with $\phi, \psi \in \mathbf{Assert}$ intended as precondition and postcondition for a program. The precondition is an assertion that is assumed to hold when the program is executed, whereas the postcondition is required to hold when its execution stops. A *Hoare triple*, written as $\{\phi\}\, C\, \{\psi\}$, expresses the fact that the program C conforms to the specification (ϕ, ψ).

Semantics. We will consider an *interpretation structure* $\mathcal{M} = (D, I)$ 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. The interpretation of expressions depends on a *state*, which is a function that maps each variable into its value. We will write $\Sigma = \mathbf{Var} \to D$ for the set of states (note that this approach extends to a multi-sorted setting by letting Σ become a *generic function space*). For $s \in \Sigma$, $s[x \mapsto a]$ will denote the state that maps x to a and any other variable y to $s(y)$. The interpretation of $e \in \mathbf{Exp}$ will be given by a function $[\![e]\!]_{\mathcal{M}} : \Sigma \to D$, and the interpretation of $b \in \mathbf{Exp^{bool}}$ will be given by $[\![b]\!]_{\mathcal{M}} : \Sigma \to \{\mathbf{F}, \mathbf{T}\}$. This reflects our assumption that an expression has a value at every state (evaluation always terminates without error) and that expression evaluation never changes the state (the language is free of *side effects*). For the interpretation of assertions we take the usual interpretation of first-order formulas, noting two facts: since assertions build on the language of program expressions their interpretation also depends on $\mathcal{M}$ (possibly extended to account for user-defined predicates and functions), and states from Σ can be used as *variable assignments* in the interpretation of assertions. The interpretation of the assertion $\phi \in \mathbf{Assert}$ is then given by $[\![\phi]\!]_{\mathcal{M}} : \Sigma \to \{\mathbf{F}, \mathbf{T}\}$, and we will write $s \models \phi$ as a shorthand for $[\![\phi]\!]_{\mathcal{M}}(s) = \mathbf{T}$. In the rest of the paper we will omit the $\mathcal{M}$ subscripts for the sake of readability; the interpretation structure will be left implicit.

1. $\langle \mathbf{skip}, s\rangle \rightsquigarrow s$
2. $\langle x := e, s\rangle \rightsquigarrow s[x \mapsto [\![e]\!](s)]$
3. if $\langle C_1, s\rangle \rightsquigarrow s'$ and $\langle C_2, s'\rangle \rightsquigarrow s''$, then $\langle C_1 \,;\, C_2, s\rangle \rightsquigarrow s''$
4. if $[\![b]\!](s) = \mathbf{T}$ and $\langle C_t, s\rangle \rightsquigarrow s'$, then $\langle \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, s\rangle \rightsquigarrow s'$
5. if $[\![b]\!](s) = \mathbf{F}$ and $\langle C_f, s\rangle \rightsquigarrow s'$, then $\langle \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, s\rangle \rightsquigarrow s'$
6. if $[\![b]\!](s) = \mathbf{T}$, $\langle C, s\rangle \rightsquigarrow s'$ and $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s'\rangle \rightsquigarrow s''$, then $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s\rangle \rightsquigarrow s''$
7. if $[\![b]\!](s) = \mathbf{F}$, then $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s\rangle \rightsquigarrow s$

Fig. 1: Evaluation semantics for *While* programs

For commands, we consider a standard operational, natural style semantics, based on a deterministic *evaluation relation* $\rightsquigarrow \subseteq \mathbf{Comm} \times \Sigma \times \Sigma$ (which again depends on an implicit interpretation of program expressions). We will write $\langle C, s\rangle \rightsquigarrow s'$ to denote the fact that if C is executed in the initial state s, then its execution terminates, and the final state is s'. The usual inductive definition of this relation is given in Figure 1.

The intuitive meaning of the triple $\{\phi\}\, C\, \{\psi\}$ is that if the program C is executed in an initial state in which the precondition ϕ is true, then either execution of C does not terminate or if it does, the postcondition ψ will be true in the final state. Because termination is not guaranteed, this is called a *partial correctness* specification. Let us define formally the validity of a Hoare triple.

Definition 1. *The Hoare triple $\{\phi\}\, C\, \{\psi\}$ is said to be* valid*, denoted $\models \{\phi\}\, C\, \{\psi\}$, whenever for all $s, s' \in \Sigma$, if $s \models \phi$ and $\langle C, s\rangle \rightsquigarrow s'$, then $s' \models \psi$.*

Hoare Calculus. Hoare [17] introduced an inference system for reasoning about Hoare triples, which we will call system H - see Figure 2 (left). Note that the system contains one rule (conseq) whose application is guarded by first-order conditions. We will consider that reasoning in this system takes place in the context of the *complete theory* $\mathrm{Th}(\mathcal{M})$ of the implicit structure $\mathcal{M}$, so that when constructing derivations in H one simply checks, when applying the (conseq) rule, whether the side conditions are elements of $\mathrm{Th}(\mathcal{M})$. We will write $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$ to denote the fact that the triple is derivable in this system with $\mathrm{Th}(\mathcal{M})$.

System H is sound w.r.t. the semantics of Hoare triples; it is also complete as long as the assertion language is sufficiently expressive (a result due to Cook [9]). One way to ensure this is to force the existence of a *strongest postcondition* for every command and assertion. Let $C \in \mathbf{Comm}$ and $\phi \in \mathbf{Assert}$, and denote by $\mathrm{post}(\phi, C)$ the set of states $\{s' \in \Sigma \mid \langle C, s\rangle \rightsquigarrow s' \text{ for some } s \in \Sigma \text{ such that } [\![\phi]\!](s) = \mathbf{T}\}$. In what follows we will assume that the assertion language $\mathbf{Assert}$ is *expressive* with respect to the command language $\mathbf{Comm}$ and interpretation structure $\mathcal{M}$, i.e., for every $\phi \in \mathbf{Assert}$ and $C \in \mathbf{Comm}$ there exists $\psi \in \mathbf{Assert}$ such that $s \models \psi$ iff $s \in \mathrm{post}(\phi, C)$ for any $s \in \Sigma$. The reader is directed to [2] for details.

Proposition 1 (Soundness of system H). *Let $C \in \mathbf{Comm}$ and $\phi, \psi \in \mathbf{Assert}$. If $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$, then $\models \{\phi\}\, C\, \{\psi\}$.*

$$\frac{}{\{\phi\}\,\mathbf{skip}\,\{\phi\}} \qquad (\text{skip}) \qquad \frac{}{\{\phi\}\,\mathbf{skip}\,\{\psi\}}\ \text{if } \phi \to \psi$$

$$\frac{}{\{\psi[e/x]\}\,x := e\,\{\psi\}} \qquad (\text{assign}) \qquad \frac{}{\{\phi\}\,x := e\,\{\psi\}}\ \text{if } \phi \to \psi[e/x]$$

$$\frac{\{\phi\}\,C_1\,\{\theta\} \qquad \{\theta\}\,C_2\,\{\psi\}}{\{\phi\}\,C_1\,;\,C_2\,\{\psi\}} \qquad (\text{seq}) \qquad \frac{\{\phi\}\,C_1\,\{\theta\} \qquad \{\theta\}\,C_2\,\{\psi\}}{\{\phi\}\,C_1\,;\,C_2\,\{\psi\}}$$

$$\frac{\{\phi \wedge b\}\,C_t\,\{\psi\} \qquad \{\phi \wedge \neg b\}\,C_f\,\{\psi\}}{\{\phi\}\,\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\,\{\psi\}} \qquad (\text{if}) \qquad \frac{\{\phi \wedge b\}\,C_t\,\{\psi\} \qquad \{\phi \wedge \neg b\}\,C_f\,\{\psi\}}{\{\phi\}\,\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\,\{\psi\}}$$

$$\frac{\{\theta \wedge b\}\,C\,\{\theta\}}{\{\theta\}\,\mathbf{while}\ b\ \mathbf{do}\ C\,\{\theta \wedge \neg b\}} \qquad (\text{while}) \qquad \frac{\{\theta \wedge b\}\,C\,\{\theta\}}{\{\phi\}\,\mathbf{while}\ b\ \mathbf{do}\,\{\theta\}\,C\,\{\psi\}}\ \text{if}\ \begin{array}{l}\phi \to \theta\ \text{ and}\\ \theta \wedge \neg b \to \psi\end{array}$$

$$\frac{\{\phi\}\,C\,\{\psi\}}{\{\phi'\}\,C\,\{\psi'\}}\ \text{if}\ \begin{array}{l}\phi' \to \phi\ \text{ and}\\ \psi \to \psi'\end{array} \qquad (\text{conseq})$$

Fig. 2: Systems H (left) and Hg (right)

Proposition 2 (Completeness of system H). *Let $C \in$ **Comm** and $\phi, \psi \in$ **Assert**. With **Assert** expressive in the above sense, if $\models \{\phi\}\,C\,\{\psi\}$, then $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi\}$.*

The sets of *variables occurring* and *assigned* in the program C will be given by $\mathsf{Vars}(C)$ and $\mathsf{Asgn}(C)$ respectively; $\mathsf{FV}(\phi)$ denotes the set of free variables occurring in ϕ (all are defined in the obvious way). We will write $\phi \# C$ to denote $\mathsf{Asgn}(C) \cap \mathsf{FV}(\phi) = \emptyset$, i.e. C does not assign variables occurring free in ϕ.

Lemma 1. *Let $\phi, \psi \in$ **Assert** and $C \in$ **Comm**, such that $\phi \# C$. If $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi\}$, then $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\phi \wedge \psi\}$.*

Goal-directed Logic. We introduce a syntactic class **AComm** of *annotated programs*, which differs from **Comm** only in the case of while commands, which are of the form **while** b **do** $\{\theta\}\,C$ where the assertion θ is a loop invariant annotation (see for instance [15]). Annotations do not affect the operational semantics. Note that for $C \in$ **AComm**, $\mathsf{Vars}(C)$ includes the free variables of the annotations in C. In what follows we will use the auxiliary function $\lfloor \cdot \rfloor :$ **AComm** $\to$ **Comm** that erases all annotations from a program (defined in the obvious way).

In Figure 2 (right) we present system Hg, a *goal-directed* version of Hoare logic for triples containing annotated programs. This system is intended for mechanical construction of derivations: loop invariants are not invented at this point but taken from the annotations, and there is no ambiguity in the choice of rule to apply, since a consequence rule is not present. The possible derivations of the same triple in Hg differ only in the intermediate assertions used. The following can be proved by induction on the derivation of $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$.

$\{n \geq 0 \land n_{aux} = n\}\, \mathsf{Fact}\, \{f = n_{aux}!\}$
(seq)

1. $\{n \geq 0 \land n_{aux} = n\}\, f := 1; i := 1\, \{n \geq 0 \land n_{aux} = n \land f = 1 \land i = 1\}$
   (seq)
   1. (assign) $\{n \geq 0 \land n_{aux} = n\}\, f := 1\, \{n \geq 0 \land n_{aux} = n \land f = 1\}$
   2. (assign) $\{n \geq 0 \land n_{aux} = n \land f = 1\}\, i := 1\, \{n \geq 0 \land n_{aux} = n \land f = 1 \land i = 1\}$
2. $\{n \geq 0 \land n_{aux} = n \land f = 1 \land i = 1\}\, \textbf{while}\ i \leq n\ \textbf{do}\, \{f = (i-1)! \land i \leq n+1 \land n_{aux} = n\}\, f := f * i; i := i + 1\, \{f = n_{aux}!\}$
   (while)
   1. $\{f = (i-1)! \land i \leq n+1 \land n_{aux} = n \land i \leq n\}\, f := f * i; i := i + 1\, \{f = (i-1)! \land i \leq n+1 \land n_{aux} = n\}$
      (seq)
      1. (assign) $\{f = (i-1)! \land i \leq n+1 \land n_{aux} = n \land i \leq n\}\, f := f * i\, \{f = (i-1)! * i \land i \leq n+1 \land n_{aux} = n \land i \leq n\}$
      2. (assign) $\{f = (i-1)! * i \land i \leq n+1 \land n_{aux} = n \land i \leq n\}\, i := i + 1\, \{f = (i-1)! \land i \leq n+1 \land n_{aux} = n\}$

Side conditions for application of the (assign) rules:

- $n \geq 0 \land n_{aux} = n \implies (n \geq 0 \land n_{aux} = n \land f = 1)[1/f]$
- $n \geq 0 \land n_{aux} = n \land f = 1 \implies (n \geq 0 \land n_{aux} = n \land f = 1 \land i = 1)[1/i]$
- $f = (i-1)! \land i \leq n+1 \land n_{aux} = n \land i \leq n \implies (f = (i-1)! * i \land i \leq n+1 \land n_{aux} = n \land i \leq n)[f * i/f]$
- $f = (i-1)! * i \land i \leq n+1 \land n_{aux} = n \land i \leq n \implies (f = (i-1)! \land i \leq n+1 \land n_{aux} = n)[i+1/i]$

Side conditions for application of the (while) rule:

- $n \geq 0 \land n_{aux} = n \land f = 1 \land i = 1 \implies f = (i-1)! \land i \leq n+1 \land n_{aux} = n$
- $f = (i-1)! \land i \leq n+1 \land n_{aux} = n \land \neg(i \leq n) \implies f = n_{aux}!$

Fig. 3: Example derivation in system Hg

Proposition 3 (Soundness of Hg). *Let $C \in$ **AComm** and $\phi, \psi \in$ **Assert**. If $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$ then $\vdash_{\mathsf{H}} \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$.*

The converse implication does not hold, since the annotated invariants may be inadequate for deriving the triple. Instead we need the following definition:

Definition 2. *Let $C \in$ **AComm** and $\phi, \psi \in$ **Assert**. We say that C is* correctly-annotated *w.r.t. (ϕ, ψ) if $\vdash_{\mathsf{H}} \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$ implies $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$.*

The following lemma states the admissibility of the consequence rule in Hg.

Lemma 2. *Let $C \in$ **AComm** and $\phi, \psi, \phi', \psi' \in$ **Assert** such that $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$, $\models \phi' \to \phi$, and $\models \psi \to \psi'$. Then $\vdash_{\mathsf{Hg}} \{\phi'\}\, C\, \{\psi'\}$.*

Consider the factorial program shown in Figure 4a. The counter i ranges from 1 to n and the accumulator f contains at each step the factorial of $i - 1$. The program is annotated with an appropriate loop invariant; it is easy to show that it is correct with respect to the specification $(n \geq 0 \land n_{aux} = n, f = n_{aux}!)$. We show in Figure 3 a derivation of this triple in system Hg. Note that the axioms $0! = 1$ and $n! = n * (n-1)!$ are required to prove the side conditions.

It is possible to write an algorithm, known as a verification conditions generator, that simply collects the side conditions of a derivation without actually constructing it. Hg is agnostic with respect to a strategy for propagating assertions, but the VCGen necessarily imposes one such strategy [15].

Exponential Explosion. To understand the exponential explosion problem mentioned in Section 1, consider a program consisting of a sequence of n conditional statements: since each such statement doubles the number of execution paths, the program has 2^n paths. Consider now the (if) rule of Hoare logic. If one uses a backward propagation strategy, one starts with a given postcondition ψ, which will be propagated through both branches of the last conditional, to produce two assertions ϕ_t, ϕ_f, both of which may contain occurrences of ψ. These will be combined in an assertion ϕ, for instance $(b \to \phi_t) \wedge (\neg b \to \phi_f)$, where ψ may occur twice. The (seq) rule will then use ϕ as postcondition for the prefix of the program, repeating the process and generating the exponential pattern. A similar exponential pattern may be generated by duplicating variables rather than assertions, in a sequence of assignment statements whose right-hand sides contain multiple occurrences of the same variable. For instance propagating backwards an assertion containing a single occurrence of z through the sequence $y := x + x\,;\, z := y + y$ produces a formula containing 4 occurrences of x.

Adaptation Incompleteness. Consider a block of code that has been proved correct with respect to a specification. Take for instance the triple $\{n \geq 0 \wedge n_{aux} = n\}\,\mathsf{Fact}\,\{f = n_{aux}!\}$. The specification makes use of an *auxiliary variable* n_{aux}. These variables do not have a special status; they are simply not used as program variables, and can be safely employed for writing specifications relating the pre-state and post-state. According to the above, the program Fact computes the factorial of the *initial* value of n. Now suppose Fact is part of a bigger program P, and one would like to establish the validity of the triple $\{n = K\}\,\mathsf{Fact}\,\{f = K!\}$, with K a positive constant. *Adaptation-completeness* would mean that one would be able to derive this from the specification of Fact without constructing a dedicated proof – indeed, it should not even be necessary to know the implementation of Fact, since it has already been proved correct. The (conseq) rule of Hoare logic is meant precisely for this, but it cannot be applied here, since both side conditions are clearly *not valid.* This shows that system H is not adaptation-complete:

$$\frac{\{n \geq 0 \wedge n_{aux} = n\}\,\mathsf{Fact}\,\{f = n_{aux}!\}}{\{n = K\}\,\mathsf{Fact}\,\{f = K!\}} \quad \text{if} \quad \begin{array}{l} n = K \to n \geq 0 \wedge n_{aux} = n \quad \text{and} \\ f = n_{aux}! \to f = K! \end{array}$$

3 Single-assignment Programs

Translation of code into Single-Assignment (SA) form has been part of the standard compilation pipeline for decades now; in such a program each variable is assigned at most once. The fragment $x := 10\,;\, x := x + 10$ could be translated as $x_1 := 10\,;\, x_2 := x_1 + 10$, using a different "version of x" variable for each assignment. In this paper we will use a dynamic notion of single-assignment (DSA) program [27], in which each variable may occur syntactically as the left-hand side of more than one assignment instruction, as long as it is not assigned more than once *in each execution.* For instance the fragment

```
$f := 1$ ;
$i := 1$ ;
while $i \leq n$ do $\{f = (i-1)! \wedge i \leq n+1$
$\wedge n_{aux} = n\}$
{
  $f := f * i$ ;
  $i := i + 1$
}
```

(a) Initial annotated program Fact

```
$f_1 := 1$ ;
$i_1 := 1$ ;
$\mathcal{I}$
while $(i_{a0} \leq n)$ do $\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1\}$
{
  $f_{a1} := f_{a0} * i_{a0}$ ;
  $i_{a1} := i_{a0} + 1$ ;
  $\mathcal{U}$
}
```

(b) With blocks converted to SA form

```
$f_1 := 1$ ;
$i_1 := 1$ ;
for $(\{i_{a0} := i_1 \,;\, f_{a0} := f_1\}, i_{a0} \leq n, \{i_{a0} := i_{a1} \,;\, f_{a0} := f_{a1}\})$ do $\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1\}$
{
  $f_{a1} := f_{a0} * i_{a0}$ ;
  $i_{a1} := i_{a0} + 1$
}
```

(c) Annotated single-assignment program $\mathsf{Fact}^{\mathsf{sa}}$

Fig. 4: Factorial example

if $x > 0$ **then** $x := x + 10$ **else skip** could be translated into DSA form as **if** $x_0 > 0$ **then** $x_1 := x_0 + 10$ **else** $x_1 := x_0$. Note the *else* branch cannot be simply **skip**, since it is necessary to have a single version variable (in this case x_1) representing x when exiting the conditional.

In the context of the guarded commands language, it has been shown [14] that VCs for *passive programs* (essentially DSA programs without loops, where assignments are replaced with assume commands) can be generated avoiding exponential explosion. However, a single-assignment language of programs with loops, tailored for verification, does not exist. In what follows we will introduce precisely such a language, based on dynamic single-assignment form.

Definition 3. *The set* **Rnm** $\subseteq$ **Comm** *of* renamings *consists of all programs of the form* $\{x_1 := y_1 \,;\, \ldots \,;\, x_n := y_n\}$ *such that all* x_i *and* y_i *are distinct. The empty renaming will be written as* **skip***.*

A renaming $\mathcal{R} = \{x_1 := y_1 \,;\, \ldots \,;\, x_n := y_n\}$ represents the finite bijection $[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$, which we will also denote by $\mathcal{R}$. We will write $\mathsf{dom}(\mathcal{R})$ and $\mathsf{rng}(\mathcal{R})$ to denote the domain and range of $\mathcal{R}$, respectively. Furthermore, $\mathcal{R}(\phi)$ will denote the assertion that results from applying the substitution $[y_1/x_1, \ldots, y_n/x_n]$ to ϕ. Also, for $s \in \Sigma$ we define the state $\mathcal{R}(s)$ as follows: $\mathcal{R}(s)(x) = s(\mathcal{R}(x))$ if $x \in \mathsf{dom}(\mathcal{R})$, and $\mathcal{R}(s)(x) = s(x)$ otherwise.

Lemma 3. *Let* $\mathcal{R} \in$ **Rnm***,* $\phi, \psi \in$ **Assert** *and* $s \in \Sigma$*.*

1. $\langle \mathcal{R}, s \rangle \rightsquigarrow \mathcal{R}(s)$
2. $[\![\mathcal{R}(\phi)]\!](s) = [\![\phi]\!](\mathcal{R}(s))$
3. $\models \{\phi\}\, \mathcal{R}\, \{\psi\}$ *iff* $\models \phi \rightarrow \mathcal{R}(\psi)$.

Proof. 1. By inspection on the evaluation relation. 2. By induction on the interpretation assertions. 3. Follows from 1 and 2. □

In a strict sense it is not possible to write iterating programs in DSA form. So what we propose here is a syntactically controlled violation of the single-assignment constraints that allows for structured reasoning. Loop bodies are still SA blocks, but two renamings, responsible for propagating the values inside, outside and between iterations, are free of single-assignment restrictions.

Definition 4. *Let* $\mathbf{AComm}^{\text{SA}}$ *be the class of* annotated single-assignment programs. *Its abstract syntax is defined by*

$$C \ ::= \ \mathbf{skip} \ | \ x := e \ | \ C \,;\, C \ | \ \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \ | \ \mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\,\{\theta\}\, C$$

where:

- $\mathbf{skip} \in \mathbf{AComm}^{\text{SA}}$
- $x := e \in \mathbf{AComm}^{\text{SA}}$ *if* $x \notin \mathsf{Vars}(e)$
- $C_1 \,;\, C_2 \in \mathbf{AComm}^{\text{SA}}$ *if* $C_1, C_2 \in \mathbf{AComm}^{\text{SA}}$ *and* $\mathsf{Vars}(C_1) \cap \mathsf{Asgn}(C_2) = \emptyset$
- $\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f \in \mathbf{AComm}^{\text{SA}}$ *if* $C_t, C_f \in \mathbf{AComm}^{\text{SA}}$ *and* $\mathsf{Vars}(b) \cap (\mathsf{Asgn}(C_t) \cup \mathsf{Asgn}(C_f)) = \emptyset$
- $\mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\,\{\theta\}\, C \in \mathbf{AComm}^{\text{SA}}$ *if* $C \in \mathbf{AComm}^{\text{SA}}$*,* $\mathcal{I}, \mathcal{U} \in \mathbf{Rnm}$*,* $\mathsf{Asgn}(\mathcal{I}) = \mathsf{Asgn}(\mathcal{U})$*,* $\mathsf{rng}(\mathcal{U}) \subseteq \mathsf{Asgn}(C)$*, and* $(\mathsf{Vars}(\mathcal{I}) \cup \mathsf{Vars}(b) \cup \mathsf{FV}(\theta)) \cap \mathsf{Asgn}(C) = \emptyset$

and Vars *and* Asgn *are extended to the* **for** *command as follows:*

- $\mathsf{Vars}(\mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\,\{\theta\}\, C) = \mathsf{Vars}(\mathcal{I}) \cup \mathsf{Vars}(b) \cup \mathsf{FV}(\theta) \cup \mathsf{Vars}(C)$
- $\mathsf{Asgn}(\mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\,\{\theta\}\, C) = \mathsf{Asgn}(\mathcal{I}) \cup \mathsf{Asgn}(C)$

Definition 4 is straightforward except in the case of loops. The initialization code $\mathcal{I}$ contains a renaming that runs exactly once, even if no iterations take place. On the other hand the code in $\mathcal{U}$ is executed after every iteration. This ensures that the variables in $\mathsf{dom}(\mathcal{U})$ (equal to $\mathsf{dom}(\mathcal{I})$) always contain the appropriate output values at the beginning of each iteration and when the loop terminates. Note that the definition of $\phi \# C$ extends to annotated programs as expected.

In Figure 4b we show again the factorial program, where we have converted the blocks to SA form (the variables occurring in the loop are signalled with an 'a' subscript for clarity, but any other fresh variables would do). The initial version variables of the loop body f_{a0} and i_{a0} are the ones used in the Boolean expression, which is evaluated at the beginning of each iteration. They are also used in the invariant annotation. We have placed in the code the required renamings $\mathcal{I}$ and $\mathcal{U}$, and it is straightforward to instantiate them. $\mathcal{I}$ should be defined as $i_{a0} := i_1 \,;\, f_{a0} := f_1$, and $\mathcal{U}$ as $i_{a0} := i_{a1} \,;\, f_{a0} := f_{a1}$. The initial version variables can be used after the loop to access the value of the counter and accumulator; so a specification for this program can be written as $(n \geq 0 \wedge n_{aux} = n, f_{a0} = n_{aux}!)$. It is now immediate to write the program with a *for* command encapsulating the structure of the loop, in accordance with Definition 4. This is shown in Figure 4c. Incidentally, note that the invariant does not contain the 'continuous' part $n_{aux} = n$ of the initial code, since it becomes unnecessary in the SA version.

The function $\mathcal{W} : \mathbf{AComm}^{\text{SA}} \rightarrow \mathbf{AComm}$ translates SA programs back to (annotated) While programs in the obvious way: $\mathcal{W}(\mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\,\{\theta\}\, C) = \mathcal{I} \,;\, \mathbf{while}\ b\ \mathbf{do}\,\{\theta\}\,\{\mathcal{W}(C) \,;\, \mathcal{U}\}$. Otherwise the function is defined as expected.

$$\text{(skip)} \qquad \frac{}{\{\phi\}\,\textbf{skip}\,\{\phi \wedge \top\}}$$

$$\text{(assign)} \qquad \frac{}{\{\phi\}\,x := e\,\{\phi \wedge x = e\}}$$

$$\text{(seq)} \qquad \frac{\{\phi\}\,C_1\,\{\phi \wedge \psi_1\} \qquad \{\phi \wedge \psi_1\}\,C_2\,\{\phi \wedge \psi_1 \wedge \psi_2\}}{\{\phi\}\,C_1\,;\,C_2\,\{\phi \wedge \psi_1 \wedge \psi_2\}}$$

$$\text{(if)} \qquad \frac{\{\phi \wedge b\}\,C_t\,\{\phi \wedge b \wedge \psi_t\} \qquad \{\phi \wedge \neg b\}\,C_f\,\{\phi \wedge \neg b \wedge \psi_f\}}{\{\phi\}\,\textbf{if}\ b\ \textbf{then}\ C_t\ \textbf{else}\ C_f\,\{\phi \wedge ((b \wedge \psi_t) \vee (\neg b \wedge \psi_f))\}}$$

$$\text{(for)} \ \frac{\{\theta \wedge b\}\,C\,\{\theta \wedge b \wedge \psi\}}{\{\phi\}\,\textbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \textbf{do}\,\{\theta\}\,C\,\{\phi \wedge \theta \wedge \neg b\}} \ \text{if} \ \begin{array}{l} \phi \rightarrow \mathcal{I}(\theta) \ \ \text{and} \\ \theta \wedge b \wedge \psi \rightarrow \mathcal{U}(\theta) \end{array}$$

Fig. 5: Inference system for annotated SA triples – System Hsa

4 Logic and Verification Conditions for SA Programs

We propose in Figure 5 an inference system for Hoare triples containing annotated SA programs. Hsa is goal-directed like system Hg but it incorporates a strategy, based on forward propagation (reminiscent of strongest postcondition computations). It is proved sound with respect to system H, and complete with respect to Hg. Note that Hsa derives triples of the form $\{\phi\}\,C\,\{\phi\wedge\psi\}$, where the program does not interfere with the truth of the precondition. For this reason we restrict our results to triples satisfying the $\phi\#C$ condition (SA translations will generate triples of this kind only).

Lemma 4. *Let $C \in \mathbf{AComm}^{\text{SA}}$ and $\phi, \psi \in \mathbf{Assert}$ such that $\phi\#C$, $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\psi\}$. Then (i) $\mathsf{FV}(\psi) \subseteq \mathsf{FV}(\phi) \cup \mathsf{Vars}(C)$ and (ii) all triples $\{\alpha\}\,C'\,\{\beta\}$ occurring in this derivation satisfy $\alpha\#C'$.*

Proof. Both are proved by induction on the structure of the derivation of $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\psi\}$. □

Proposition 4 (Soundness of system Hsa). *Let $C \in \mathbf{AComm}^{\text{SA}}$ and ϕ, $\psi' \in \mathbf{Assert}$ such that $\phi\#C$. If $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\phi\wedge\psi'\}$, then $\vdash_{\mathsf{H}} \{\phi\}\,\lfloor\mathcal{W}(C)\rfloor\,\{\phi\wedge\psi'\}$.*

Proof. By induction on the derivation of $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\phi\wedge\psi'\}$, using Lemma 4 and induction hypotheses. We show the interesting case, where the last rule applied is (for). Assume the last step is

$$\frac{\{\theta \wedge b\}\,C\,\{\theta \wedge b \wedge \psi\}}{\{\phi\}\,\textbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \textbf{do}\,\{\theta\}\,C\,\{\phi \wedge \theta \wedge \neg b\}} \ \text{with} \ \begin{array}{l} \phi \rightarrow \mathcal{I}(\theta) \ \ \text{and} \\ \theta \wedge b \wedge \psi \rightarrow \mathcal{U}(\theta) \end{array}$$

By Lemma 4, we have that $(\theta \wedge b)\#C$. So, by induction hypothesis, we have $\vdash_{\mathsf{H}} \{\theta \wedge b\}\,\lfloor\mathcal{W}(C)\rfloor\,\{\theta \wedge b \wedge \psi\}$. From the validity of the side conditions, by

Lemma 3 and completeness of H, we have $\vdash_{\mathsf{H}} \{\theta \wedge b \wedge \psi\}\, \mathcal{U}\, \{\theta\}$ and $\vdash_{\mathsf{H}} \{\phi\}\, \mathcal{I}\, \{\theta\}$. Now applying sequentially the rules (seq), (while) and again (seq), we get $\vdash_{\mathsf{H}} \{\phi\}\, \mathcal{I}\, ;\, \textbf{while}\, b\, \textbf{do}\, \{\lfloor \mathcal{W}(C) \rfloor\, ;\, \mathcal{U}\}\, \{\theta \wedge \neg b\}$. Hence, by definition of $\mathcal{W}$ and Lemma 1, we have $\vdash_{\mathsf{H}} \{\phi\}\, \lfloor \mathcal{W}(\textbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \textbf{do}\, \{\theta\}\, C) \rfloor\, \{\phi \wedge \theta \wedge \neg b\}$. □

Proposition 5 (Completeness of system Hsa). *Let $C \in \mathbf{AComm}^{\mathrm{SA}}$ and ϕ, $\psi \in \mathbf{Assert}$ such that $\phi \# C$ and $\vdash_{\mathsf{Hg}} \{\phi\}\, \mathcal{W}(C)\, \{\psi\}$. Then $\vdash_{\mathsf{Hsa}} \{\phi\}\, C\, \{\phi \wedge \psi'\}$ for some $\psi' \in \mathbf{Assert}$ such that $\models \phi \wedge \psi' \rightarrow \psi$.*

Proof. By induction on the structure of C. Assume $\phi \# C$ and $\vdash_{\mathsf{Hg}} \{\phi\}\, \mathcal{W}(C)\, \{\psi\}$.

- Case $C \equiv x := e$, we must have $\models \phi \rightarrow \psi[e/x]$. Since $x \notin (\mathsf{FV}(e) \cup \mathsf{FV}(\phi))$, it follows that $\models \phi \wedge x = e \rightarrow \psi$. As $\vdash_{\mathsf{Hsa}} \{\phi\}\, x := e\, \{\phi \wedge x = e\}$ we are done.
- Case $C \equiv C_1; C_2$, we must have for some $\gamma \in \mathbf{Assert}$ $\vdash_{\mathsf{Hg}} \{\phi\}\, \mathcal{W}(C_1)\, \{\gamma\}$ and $\vdash_{\mathsf{Hg}} \{\gamma\}\, \mathcal{W}(C_2)\, \{\psi\}$. Since $\phi \# C_1; C_2$ we have $\phi \# C_1$. Hence by induction hypothesis we have $\vdash_{\mathsf{Hsa}} \{\phi\}\, C_1\, \{\phi \wedge \gamma'\}$ for some $\gamma' \in \mathbf{Assert}$ such that $\models \phi \wedge \gamma' \rightarrow \gamma$. Therefore, by Lemma 2, $\vdash_{\mathsf{Hg}} \{\phi \wedge \gamma'\}\, \mathcal{W}(C_2)\, \{\psi\}$. From Lemma 4 we have that $\mathsf{FV}(\phi \wedge \gamma') \subseteq \mathsf{FV}(\phi) \cup \mathsf{Vars}(C_1)$, and thus $(\phi \wedge \gamma') \# C_2$. Hence by induction hypothesis $\vdash_{\mathsf{Hsa}} \{\phi \wedge \gamma'\}\, C_2\, \{\phi \wedge \gamma' \wedge \psi'\}$ for some $\psi' \in \mathbf{Assert}$ such that $\models \phi \wedge \gamma' \wedge \psi' \rightarrow \psi$. Applying rule (seq) we then get $\vdash_{\mathsf{Hsa}} \{\phi\}\, C_1\, ;\, C_2\, \{\phi \wedge \gamma' \wedge \psi'\}$.
- Case $C \equiv \textbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \textbf{do}\, \{\theta\}\, C_t$, we must have, for some $\gamma \in \mathbf{Assert}$, that $\vdash_{\mathsf{Hg}} \{\phi\}\, \mathcal{I}\, \{\theta\}$, $\vdash_{\mathsf{Hg}} \{\theta \wedge b\}\, \mathcal{W}(C_t)\, \{\gamma\}$, $\vdash_{\mathsf{Hg}} \{\gamma\}\, \mathcal{U}\, \{\theta\}$, and $\models \theta \wedge \neg b \rightarrow \psi$. We have that $(\theta \wedge b) \# C_t$, so it follows by induction hypothesis that $\vdash_{\mathsf{Hsa}} \{\theta \wedge b\}\, C_t\, \{\theta \wedge b \wedge \gamma'\}$, for some $\gamma' \in \mathbf{Assert}$ and $\models \theta \wedge b \wedge \gamma' \rightarrow \gamma$. Therefore, by Lemma 2, $\vdash_{\mathsf{Hg}} \{\theta \wedge b \wedge \gamma'\}\, \mathcal{U}\, \{\theta\}$. Since Hg is sound, by Lemma 3, it follows that $\models \phi \rightarrow \mathcal{I}(\theta)$ and $\models \theta \wedge b \wedge \gamma' \rightarrow \mathcal{U}(\theta)$, which allow us to apply rule (for) and get the conclusion $\vdash_{\mathsf{Hsa}} \{\phi\}\, \textbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \textbf{do}\, \{\theta\}\, C\, \{\phi \wedge \theta \wedge \neg b\}$.

The remaining cases are routine. □

All the rules of system Hsa propagate the precondition ϕ forward. Note that in the (for) rule this happens even though ϕ is not implied by the annotated loop invariant. Observe also how in this same rule we reason structurally about the body of the loop (an SA piece of code), with the renamings applied to the invariant in the side conditions.

Figure 6 shows an example of a Hsa derivation, where $\mathsf{Fact}^{\mathsf{sa}}$ is the factorial single-assignment program of Figure 4c. The assertion $n \geq 0 \wedge n_{aux} = n$ is used as precondition. Note that the application of the (for) rule introduces two side conditions, which are both valid. The derivation generates a unique postcondition for the program, with the given precondition. Other valid triples with the same precondition may be obtained by weakening this postcondition, following Proposition 5. For the triple $\{n \geq 0 \wedge n_{aux} = n\}\, \mathsf{Fact}\, \{f_{a0} = n_{aux}!\}$ we would check the validity of:

$$n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1 \wedge f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge \neg(i_{a0} \leq n) \ \rightarrow\ f_{a0} = n_{aux}!$$

A set of verification conditions for a triple $\{\phi\}\, C\, \{\psi\}$ can be obtained from a candidate derivation of a triple of the form $\{\phi\}\, C\, \{\phi \wedge \psi'\}$ in system Hsa. The VCs

$\{n \geq 0 \wedge n_{aux} = n\}$
$\mathsf{Fact}^{\mathsf{sa}}$
$\{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1 \wedge f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge \neg(i_{a0} \leq n)\}$
(seq)

1. $\{n \geq 0 \wedge n_{aux} = n\}\, f_1 := 1; i_1 := 1\, \{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1\}$
   (seq)
   1. (assign) $\{n \geq 0 \wedge n_{aux} = n\}\, f_1 := 1\, \{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1\}$
   2. (assign) $\{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1\}\, i_1 := 1\, \{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1\}$
2. $\{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1\}$
   $\mathbf{for}\ (\{i_{a0} := i_1; f_{a0} := f_1\}, i_{a0} \leq n, \{i_{a0} := i_{a1}; f_{a0} := f_{a1}\})\ \mathbf{do}\, \{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1\}\, \{f_{a1} := f_{a0} * i_{a0}; i_{a1} := i_{a0} + 1\}$
   $\{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1 \wedge f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge \neg(i_{a0} \leq n)\}$
   (for)
   1. $\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n\}$
      $f_{a1} := f_{a0} * i_{a0}; i_{a1} := i_{a0} + 1$
      $\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n \wedge f_{a1} = f_{a0} * i_{a0} \wedge i_{a1} = i_{a0} + 1\}$
      (seq)
      1. (assign) $\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n\}\, f_{a1} := f_{a0} * i_{a0}\, \{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n \wedge f_{a1} = f_{a0} * i_{a0}\}$
      2. (assign) $\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n \wedge f_{a1} = f_{a0} * i_{a0}\}\, i_{a1} := i_{a0} + 1\, \{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n \wedge f_{a1} = f_{a0} * i_{a0} \wedge i_{a1} = i_{a0} + 1\}$

Side conditions for application of the (for) rule:

- $n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1 \rightarrow f_1 = (i_1 - 1)! \wedge i_1 \leq n + 1$
- $f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n \wedge f_{a1} = f_{a0} * i_{a0} \wedge i_{a1} = i_{a0} + 1 \rightarrow f_{a1} = (i_{a1} - 1)! \wedge i_{a1} \leq n + 1$

Fig. 6: Example derivation in system Hsa

are the side conditions introduced by the (for) rule, together with $\phi \wedge \psi' \rightarrow \psi$: the triple is valid if and only if all these VCs are valid. It is possible to calculate the VCs and the formula ψ' without explicitly constructing the derivation. The following function does precisely this.

Definition 5 (Verification Conditions Generator). *The VCGen function* $\mathsf{VC} : \mathbf{Assert} \times \mathbf{AComm}^{\mathrm{SA}} \rightarrow \mathbf{Assert} \times \mathcal{P}(\mathbf{Assert})$ *is defined as follows:*

$$\begin{aligned}
\mathsf{VC}(\phi, \mathbf{skip}) &= (\top, \emptyset) \\
\mathsf{VC}(\phi, x := e) &= (x = e, \emptyset) \\
\mathsf{VC}(\phi, C_1\, ;\, C_2) &= (\psi_1 \wedge \psi_2, \Gamma_1 \cup \Gamma_2) \\
&\quad \textit{where } (\psi_1, \Gamma_1) = \mathsf{VC}(\phi, C_1) \textit{ and} \\
&\quad \phantom{\textit{where }} (\psi_2, \Gamma_2) = \mathsf{VC}(\phi \wedge \psi_1, C_2) \\
\mathsf{VC}(\phi, \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f) &= ((b \wedge \psi_t) \vee (\neg b \wedge \psi_f), \Gamma_t \cup \Gamma_f) \\
&\quad \textit{where } (\psi_t, \Gamma_t) = \mathsf{VC}(\phi \wedge b, C_t) \textit{ and} \\
&\quad \phantom{\textit{where }} (\psi_f, \Gamma_f) = \mathsf{VC}(\phi \wedge \neg b, C_f) \\
\mathsf{VC}(\phi, \mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\ \{\theta\}\ C) &= (\theta \wedge \neg b, \Gamma \cup \{\phi \rightarrow \mathcal{I}(\theta), \theta \wedge b \wedge \psi \rightarrow \mathcal{U}(\theta)\}) \\
&\quad \textit{where } (\psi, \Gamma) = \mathsf{VC}(\theta \wedge b, C)
\end{aligned}$$

Let $(\psi', \Gamma) = \mathsf{VC}(\phi, C)$. The verification conditions of C with the precondition ϕ are given by the set Γ, and the formula ψ' approximates (since it relies on loop invariants) a logical encoding of the program; it is clear from the definition that ψ' does not depend on the formula ϕ. The VCs of a Hoare triple $\{\phi\}\, C\, \{\psi\}$ are then given by $\Gamma \cup \{\phi \wedge \psi' \rightarrow \psi\}$.

Proposition 6. *Let* $C \in \mathbf{AComm}^{\mathrm{SA}}$, $\phi, \psi', \psi'' \in \mathbf{Assert}$ *and* $\Gamma \subseteq \mathbf{Assert}$, *such that* $(\psi', \Gamma) = \mathsf{VC}(\phi, C)$. *Then:*

1. *If* $\models \Gamma$*, then* $\vdash_{\mathsf{Hsa}} \{\phi\}\, C\, \{\phi \wedge \psi'\}$.
2. *If* $\vdash_{\mathsf{Hsa}} \{\phi\}\, C\, \{\phi \wedge \psi''\}$*, then* $\models \Gamma$ *and* $\psi'' \equiv \psi'$.

Proof. 1. By induction on the structure of C. 2. By induction on the derivation of $\vdash_{\mathsf{Hsa}} \{\phi\}\, C\, \{\phi \wedge \psi''\}$. □

The reader may check that for our factorial example we have $\mathsf{VC}(n \geq 0 \wedge n_{aux} = n, \mathsf{Fact}) = (f_1 = 1 \wedge i_1 = 1 \wedge f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n+1 \wedge \neg(i_{a0} \leq n), \{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1 \rightarrow f_1 = (i_1 - 1)! \wedge i_1 \leq n+1, f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n+1 \wedge i_{a0} \leq n \wedge f_{a1} = f_{a0} * i_{a0} \wedge i_{a1} = i_{a0} + 1 \rightarrow f_{a1} = (i_{a1} - 1)! \wedge i_{a1} \leq n+1\})$, in accordance with the derivation of Figure 6.

Consider the calculation of $\mathsf{VC}(\phi, \{\mathbf{if}\ b\ \mathbf{then}\, C_t\, \mathbf{else}\, C_f\}; C_2)$. The recursive call on C_2 will be $\mathsf{VC}(\phi \wedge ((b \wedge \psi_t) \vee (\neg b \wedge \psi_f)), C_2)$, where ψ_t, ψ_f do not depend on ϕ. The resulting VCs avoid the exponential pattern described in Section 2, since a single copy of the precondition ϕ is propagated to C_2. In fact the size of the VCs is *quadratic* on the size of the program. It is clear from the $\mathsf{VC}(\phi, C_1\,;\, C_2)$ clause of the definition that the propagated precondition ϕ is duplicated, with one copy used to generate VCs for C_1, and another propagated to C_2 together with the encoding of C_1. Now observe that each loop in the program generates two VCs, one corresponding to the initialization of the invariant ($\phi \rightarrow \mathcal{I}(\theta)$), and another to its preservation. The size of loop preservation VCs depends only on the size of the loop's body, but initialization conditions contain an encoding of the prefix of the program leading to the loop (propagated in the ϕ parameter), so they have size linear in the size of that prefix. The worst case occurs for a program consisting in a sequence of n loops: the i^{th} loop will generate an initialization VC of size $\mathcal{O}(i)$, so the total size of the VCs is $\mathcal{O}(n^2)$.

This VCGen can in fact be simplified, in a way that potentially decreases the size of the VCs. We have seen that the propagation of assertions (using the ϕ parameter) is a potential source of formula duplication, but in fact the ϕ parameter can be eliminated. For this, the algorithm must now return a triple (ψ, γ, Γ) containing, in addition to an encoding ψ of the program and a set Γ of VCs, the VC that is currently being constructed (whereas the conditions in Γ have already been fully generated, inside inner loops of the current block). The simplified VCGen highlights the fundamental fact that VC generation for SA programs is not a matter of directed 'propagation' of assertions, either forward or backward: it suffices to perform a single program traversal collecting pieces of information along the way, and conveniently structuring them:

$$
\begin{aligned}
\mathsf{VC_L}(\mathbf{skip}) &= (\top, \top, \emptyset) \\
\mathsf{VC_L}(x := e) &= (x = e, \top, \emptyset) \\
\mathsf{VC_L}(C_1\,;\, C_2) &= (\psi_1 \wedge \psi_2, \gamma_1 \wedge (\psi_1 \rightarrow \gamma_2), \Gamma_1 \cup \Gamma_2) \\
&\quad\ \text{where } (\psi_1, \gamma_1, \Gamma_1) = \mathsf{VC_L}(C_1) \ \text{ and } \ (\psi_2, \gamma_2, \Gamma_2) = \mathsf{VC_L}(C_2) \\
\mathsf{VC_L}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f) &= ((b \wedge \psi_t) \vee (\neg b \wedge \psi_f), (b \rightarrow \gamma_t) \wedge (\neg b \rightarrow \gamma_f), \Gamma_t \cup \Gamma_f) \\
&\quad\ \text{where } (\psi_t, \gamma_t, \Gamma_t) = \mathsf{VC_L}(C_t) \text{ and } (\psi_f, \gamma_f, \Gamma_f) = \mathsf{VC_L}(C_f) \\
\mathsf{VC_L}(\mathbf{for}\,(\mathcal{I}, b, \mathcal{U})\,\mathbf{do}\,\{\theta\}\, C) &= (\theta \wedge \neg b, \mathcal{I}(\theta), \{\theta \wedge b \rightarrow \gamma \wedge (\psi \rightarrow \mathcal{U}(\theta))\} \cup \Gamma) \\
&\quad\ \text{where } (\psi, \gamma, \Gamma) = \mathsf{VC_L}(C)
\end{aligned}
$$

Let $(\psi'_l, \gamma_l, \Gamma_l) = \mathsf{VC_L}(C)$ and $(\psi', \Gamma) = \mathsf{VC}(\phi, C)$. Clearly ψ'_l and ψ' are the same, and it can be proved by induction that $\bigwedge \Gamma \Leftrightarrow \bigwedge \Gamma_l \wedge (\phi \to \gamma_l)$. The VCs of a Hoare triple $\{\phi\}\, C\, \{\psi\}$ are then given by $\Gamma_l \cup \{\phi \to \gamma_l \wedge (\psi'_l \to \psi)\}$.

With respect to VC size, note that this VCGen joins the initialization VCs of all the top-level loops in each sequence in a single condition. Instead of replicating prefixes of the program for each VC, a single formula is generated, that will be valid only when all initialization conditions hold. For left-associative sequences the size of this formula may still be quadratic, since the sequence clause duplicates the program formula ψ_1 of C_1; however, if sequences are represented in a right-associative way (a reasonable assumption for an intermediate language), the size of the resulting VCs is *linear in the size of* C in the worst-case.

5 Program Verification Using Intermediate SA Form

We will now put up a framework for the verification of annotated While programs, based on their translation to single-assignment form and the subsequent generation of compact verification conditions from this intermediate code.

The translation into SA form will operate at the level of Hoare triples, rather than of isolated annotated programs. Such a translation must of course abide by the syntactic restrictions of $\mathbf{AComm}^{\mathrm{SA}}$ (as illustrated by the factorial example), with additional requirements of a semantic nature. In particular, the translation will annotate the SA program with loop invariants (produced from those contained in the original program), and Hg-derivability guided by these annotations must be preserved. On the other hand, the translation must be sound: it will not translate invalid triples into valid ones. These requirements are expressed by translating back to While programs.

Definition 6 (SA translation). *A function* $\mathcal{T} : \mathbf{Assert} \times \mathbf{AComm} \times \mathbf{Assert} \to \mathbf{Assert} \times \mathbf{AComm}^{\mathrm{SA}} \times \mathbf{Assert}$ *is said to be a* single-assignment translation *if when* $\mathcal{T}(\phi, C, \psi) = (\phi', C', \psi')$ *we have* $\phi' \# C'$*, and both the following hold:*

1. *If* $\models \{\phi'\}\, \lfloor \mathcal{W}(C') \rfloor\, \{\psi'\}$*, then* $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$*.*
2. *If* $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$*, then* $\vdash_{\mathsf{Hg}} \{\phi'\}\, \mathcal{W}(C')\, \{\psi'\}$*.*

The following results establish that translating annotated programs to an intermediate SA form before generating VCs results in a sound and complete technique for deductive verification.

Proposition 7 (Soundness of verification technique). *Let* $C \in \mathbf{AComm}$*,* $C' \in \mathbf{AComm}^{\mathrm{SA}}$*,* $\phi, \phi', \psi, \psi', \gamma \in \mathbf{Assert}$ *and* $\Gamma \subseteq \mathbf{Assert}$*, such that* $(\phi', C', \psi') = \mathcal{T}(\phi, C, \psi)$ *for some SA translation* $\mathcal{T}$*, and* $(\gamma, \Gamma) = \mathsf{VC}(\phi', C')$*.*
If $\models \Gamma$*,* $\phi' \wedge \gamma \to \psi'$ *then* $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$*.*

Proof. From Proposition 6(1) we have $\vdash_{\mathsf{Hsa}} \{\phi'\}\, C'\, \{\phi' \wedge \gamma\}$ and from Definition 6 we have $\phi' \# C'$. Thus Proposition 4 applies yielding $\vdash_{\mathsf{H}} \{\phi'\}\, \lfloor \mathcal{W}(C') \rfloor\, \{\phi' \wedge \gamma\}$. From soundness of H, and because $\models \phi' \wedge \gamma \to \psi'$, it follows that $\models \{\phi'\}\, \lfloor \mathcal{W}(C') \rfloor\, \{\psi'\}$. Finally, by Definition 6, we have $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$. □

Proposition 8 (Completeness of verification technique). *Let $C \in$* **AComm***, $C' \in$* $\mathbf{AComm}^{\mathrm{SA}}$*, $\phi, \phi', \psi, \psi', \gamma \in$* **Assert** *and $\Gamma \subseteq$* **Assert** *such that $(\phi', C', \psi') = \mathcal{T}(\phi, C, \psi)$ for some SA translation $\mathcal{T}$, and $(\gamma, \Gamma) = \mathsf{VC}(\phi', C')$. If $\models \{\phi\} \lfloor C \rfloor \{\psi\}$ and C is correctly-annotated w.r.t. (ϕ, ψ), then $\models \Gamma$, $\phi' \wedge \gamma \to \psi'$.*

Proof. First note that by completeness of system H we have $\vdash_{\mathsf{H}} \{\phi\} \lfloor C \rfloor \{\psi\}$. By Definitions 2 and 6 it follows that $\vdash_{\mathsf{Hg}} \{\phi\}\, C \,\{\psi\}$ and $\vdash_{\mathsf{Hg}} \{\phi'\}\, \mathcal{W}(C') \,\{\psi'\}$. The latter definition implies that $\phi' \# C'$, and by Proposition 5 $\vdash_{\mathsf{Hsa}} \{\phi'\}\, C' \,\{\phi' \wedge \psi''\}$ for some $\psi'' \in$ **Assert** such that $\models \phi' \wedge \psi'' \to \psi'$. Proposition 6(2) then gives us $\models \Gamma$ and $\psi'' \equiv \gamma$, which concludes the proof. □

An example of a detailed translation can be found in Appendix A, together with the proof that it complies to Definition 6.

6 Adaptation Completeness of SA Program Logic

Let (ϕ, ψ) and (ϕ', ψ') be specifications, and assume that (ϕ, ψ) is satisfiable (there exists some program that is correct w.r.t. it). Suppose now that C is a program such that if the Hoare triple $\{\phi\}\, C \,\{\psi\}$ is valid then so is $\{\phi'\}\, C \,\{\psi'\}$. An inference system for Hoare logic is said to be *adaptation-complete* if whenever this happens, then $\{\phi'\}\, C \,\{\psi'\}$ is derivable in that system from the triple $\{\phi\}\, C \,\{\psi\}$.

Adaptation is closely linked to the existence of a *consequence rule* that dictates when a triple is derivable in one step from another triple containing the same program. Adaptation is by design entirely absent from goal-directed systems like Hg or Hsa, which have no consequence rule. System H is capable of adaptation, but not in a complete way. An example of this was already seen at the end of Section 2, involving the use of auxiliary variables. For an even simpler example of how adaptation fails in system H, consider the triple $\{x > 0\}\, P \,\{y = x\}$ where x is now a program variable, used outside P, but *not assigned in P*. Again let K be some positive constant. Clearly if the triple is valid then so is $\{x = K\}\, P \,\{y = K\}$, since the value of x is preserved. However, attempting to apply the consequence rule would yield the following, where the first side condition is valid, but the second is invalid

$$\frac{\{x > 0\}\, P \,\{y = x\}}{\{x = K\}\, P \,\{y = K\}} \quad \text{if} \quad \begin{array}{l} x = K \to x > 0 \quad \text{and} \\ y = x \to y = K \end{array}$$

The problem of adaptation was raised by the study of complete extensions of Hoare logic for reasoning about recursive procedures. The initial proposal by Hoare [18] was to derive a triple concerning a procedure call by assuming that same triple as a hypothesis when reasoning about the procedure's body:

$$\frac{\{\phi\}\, \mathbf{call\, p} \,\{\psi\} \vdash \{\phi\}\, \mathbf{body(p)} \,\{\psi\}}{\{\phi\}\, \mathbf{call\, p} \,\{\psi\}}$$

(assuming that an identity axiom is present, and system H rules are lifted to work with sequents). It was soon discovered that, in the presence of auxiliary variables in the procedure's specification, the resulting system turned out to be incomplete, and the reason for this was the failure to handle the adaptation of the procedure's specification to the local context of the recursive call.

One solution for this problem was to introduce additional structural rules [1], but Kleymann [20] has shown that the adaptation problem is orthogonal to the handling of recursive procedures: if the base system is made adaptation-complete, then Hoare's rule for recursive procedure calls is sufficient to achieve a Cook-complete system, with no need for further structural rules.

Kleymann obtains an adaptation-complete inference system for Hoare logic by proposing the following consequence rule, whose side condition is a meta-level formula with quantification over states/variable assignments:

$$(\text{conseq}_K) \qquad \frac{\{\phi\}\, C\, \{\psi\}}{\{\phi'\}\, C\, \{\psi'\}} \qquad \begin{array}{l} \text{if } \ \forall Z.\forall\sigma.[\![\phi']\!](Z,\sigma) \to \\ \qquad \forall\tau.(\forall Z_1.[\![\phi]\!](Z_1,\sigma) \to [\![\psi]\!](Z_1,\tau)) \\ \qquad \to [\![\psi']\!](Z,\tau) \end{array}$$

$[\![\phi']\!](Z,\sigma)$ denotes the truth value of ϕ' in the state (Z,σ), partitioned between auxiliary (Z) and program (σ) variables. As it is, (conseq_K) cannot be handled directly by an SMT solver, since the condition is not a first-order formula.

We will show that reasoning with single-assignment programs is advantageous from the point of view of adaptation: our Hsa system will be made adaptation-complete by adding a rule with a simple syntactic side condition. We start with a result showing that the side condition of a consequence rule that always leads to adaptation-completeness, in general terms, turns out to be the result of stripping away the states and quantifiers in the side condition of (conseq_K) above.

Lemma 5. *Let $\phi, \phi', \psi, \psi' \in$ **Assert**. If there exists at least one program $C_0 \in$ **Comm** such that $\models \{\phi\}\, C_0\, \{\psi\}$, and for arbitrary C one has that $\models \{\phi\}\, C\, \{\psi\}$ implies $\models \{\phi'\}\, C\, \{\psi'\}$, then it must be the case that $\models \phi' \to (\phi \to \psi) \to \psi'$.*

Proof. We assume $\not\models \phi' \to (\phi \to \psi) \to \psi'$, i.e. there exists a state s_0 such that $s_0 \models \phi'$, $s_0 \models \phi \to \psi$, and $s_0 \not\models \psi'$. To show that in this context $\models \{\phi'\}\, C\, \{\psi'\}$ does not follow from $\models \{\phi\}\, C\, \{\psi\}$ for arbitrary C, we construct a particular program C_1 with the following behavior: $\langle C_1, s_0\rangle \rightsquigarrow s_0$ and for $s \neq s_0$,$\langle C_1, s\rangle \rightsquigarrow s'$ whenever $\langle C_0, s\rangle \rightsquigarrow s'$. To see that $\{\phi\}\, C_1\, \{\psi\}$ is a valid triple, observe that if $s_0 \models \phi$ and C_1 is executed in state s_0 we will have $s_0 \models \psi$ since $s_0 \models \phi \to \psi$, and for other executions we note that $\models \{\phi\}\, C_0\, \{\psi\}$. The triple $\{\phi'\}\, C_1\, \{\psi'\}$ is however not valid, since $s_0 \models \phi'$, but $\langle C_1, s_0\rangle \rightsquigarrow s_0$ and $s_0 \not\models \psi'$. □

The problem is that a consequence rule with side condition $\phi' \to (\phi \to \psi) \to \psi'$ would not be sound. But it is sound for triples satisfying the simple syntactic restriction that free variables of the precondition are not assigned in the program.

Lemma 6. *Let $C \in$ **Comm** and $\phi \in$ **Assert**. If $\phi \# C$ and $\langle C, s\rangle \rightsquigarrow s'$, then $[\![\phi]\!](s) = [\![\phi]\!](s')$.*

Proof. Since $\phi \# C$, $s(x) = s'(x)$ for every $x \in \mathsf{FV}(\phi)$. Hence, $[\![\phi]\!](s) = [\![\phi]\!](s')$. □

Lemma 7. *Let* $C \in \mathbf{Comm}$ *and* $\phi, \phi', \psi, \psi' \in \mathbf{Assert}$*, such that* $\phi \# C$ *and* $\phi' \# C$*. If* $\models \{\phi\}\, C\, \{\psi\}$ *and* $\models \phi' \rightarrow (\phi \rightarrow \psi) \rightarrow \psi'$*, then* $\models \{\phi'\}\, C\, \{\psi'\}$*.*

Proof. Assume $s \models \phi'$ and $\langle C, s\rangle \rightsquigarrow s'$. Since $\phi' \# C$, by Lemma 6, we get $s' \models \phi'$. We also have $s' \models \phi \rightarrow \psi$ because, if $s' \models \phi$, then $s \models \phi$ (by Lemma 6, since $\phi \# C$) so, as $\models \{\phi\}\, C\, \{\psi\}$, we get $s' \models \psi$. Now, $s' \models \psi'$ follows directly from $\models \phi' \rightarrow (\phi \rightarrow \psi) \rightarrow \psi'$, $s' \models \phi'$ and $s' \models \phi \rightarrow \psi$. □

Recall from Lemma 4 that Hsa derivations consist entirely of triples $\{\phi\}\, C\, \{\psi\}$ satisfying the $\phi \# C$ condition, which means that an adaptation rule with the side condition given above can be naturally incorporated in the system. We must however be careful to ensure that the new rule *preserves* Lemma 4; in particular, the postcondition ψ' should not contain free occurrences of variables not occurring either in the program or free in the precondition ϕ'. The following result will allow us to eliminate these free occurrences.

Lemma 8. *Let* $C \in \mathbf{Comm}$*,* $\phi, \psi \in \mathbf{Assert}$ *and* $x \in \mathbf{Var}$*, such that* $x \notin \mathsf{FV}(\phi) \cup \mathsf{Vars}(C)$*. If* $\models \{\phi\}\, C\, \{\psi\}$ *then* $\models \{\phi\}\, C\, \{\forall\, x.\, \psi\}$*.*

Proof. Assume $s \models \phi$ and $\langle C, s\rangle \rightsquigarrow s'$. As $x \notin \mathsf{FV}(\phi) \cup \mathsf{Vars}(C)$, for every $a \in D$, $s[x \mapsto a] \models \phi$ and $\langle C, s[x \mapsto a]\rangle \rightsquigarrow s'[x \mapsto a]$. Since $\models \{\phi\}\, C\, \{\psi\}$, it follows that $s'[x \mapsto a] \models \psi$. Hence, $s' \models \forall\, x.\, \psi$.

Let Hsa^{+} be the inference system consisting of all the rules of system Hsa together with the following rule:

$$(\text{conseq}_a) \qquad \frac{\{\phi\}\, C\, \{\phi \wedge \psi\}}{\{\phi'\}\, C\, \{\phi' \wedge (\forall\, \boldsymbol{x}.\, \phi \rightarrow \psi)\}} \quad \text{if} \quad \begin{array}{l} \phi \# C \text{ and} \\ \boldsymbol{x} = \mathsf{FV}(\phi) \backslash (\mathsf{FV}(\phi') \cup \mathsf{Vars}(C)) \end{array}$$

Recall that Hsa is a forward propagation system, so the rule will be applied when we reach C with the propagated precondition ϕ' (in which case Lemma 4 ensures that $\phi' \# C$ holds). The rule will produce a postcondition not directly by propagating ϕ' through the structure of C, but instead by adapting the triple $\{\phi\}\, C\, \{\phi \wedge \psi\}$. The conditions ϕ and ψ may well contain occurrences of variables not occurring either in C or free in ϕ', but the quantification ensures that Lemma 4 remains valid in system Hsa^{+}. Note that the lemma guarantees that $\mathsf{FV}(\psi) \subseteq \mathsf{FV}(\phi) \cup \mathsf{Vars}(C)$, so $\mathsf{FV}(\psi)$ does not need to be included in $\boldsymbol{x}$.

System Hsa^{+} is not goal-directed, but it is still a forward-propagation system (the postcondition is the strongest allowed by Lemma 7).

Proposition 9 (Soundness of Hsa^{+}). *Let* $C \in \mathbf{AComm}^{\mathrm{SA}}$ *and* ϕ*,* $\psi' \in \mathbf{Assert}$ *such that* $\phi \# C$ *and* $\vdash_{\mathsf{Hsa}^{+}} \{\phi\}\, C\, \{\phi \wedge \psi'\}$*. Then* $\vdash_{\mathsf{H}} \{\phi\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\phi \wedge \psi'\}$*.*

Proof. The proof, by induction on the structure of the derivation of $\vdash_{\mathsf{Hsa}^{+}} \{\phi\}\, C\, \{\phi \wedge \psi'\}$, extends the proof of Proposition 4 with the (conseq_a) rule case. Assume the last step is

$$\frac{\{\phi_1\}\, C\, \{\phi_1 \wedge \psi_1\}}{\{\phi\}\, C\, \{\phi \wedge (\forall\, \boldsymbol{x}.\, \phi_1 \rightarrow \psi_1)\}} \quad \text{with} \quad \phi_1 \# C \text{ and } \boldsymbol{x} = \mathsf{FV}(\phi_1) \backslash (\mathsf{FV}(\phi) \cup \mathsf{Vars}(C))$$

By induction hypothesis we have $\vdash_{\mathsf{H}} \{\phi_1\} \lfloor \mathcal{W}(C) \rfloor \{\phi_1 \wedge \psi_1\}$ and since H is sound it follows that $\models \{\phi_1\} \lfloor \mathcal{W}(C) \rfloor \{\phi_1 \wedge \psi_1\}$. As $\models \phi \rightarrow (\phi_1 \rightarrow \phi_1 \wedge \psi_1) \rightarrow \phi \wedge (\phi_1 \rightarrow \psi_1)$, we get $\models \{\phi\} \lfloor \mathcal{W}(C) \rfloor \{\phi \wedge (\phi_1 \rightarrow \psi_1)\}$ by Lemma 7. Now note that $\boldsymbol{x} \cap (\mathsf{FV}(\phi) \cup \mathsf{Vars}(C)) = \emptyset$, thus Lemma 8 can be applied, and it follows that $\models \{\phi\} \lfloor \mathcal{W}(C) \rfloor \{\phi \wedge (\forall \boldsymbol{x}.\, \phi_1 \rightarrow \psi_1)\}$. Finally, by completeness of H we obtain $\vdash_{\mathsf{H}} \{\phi\} \lfloor \mathcal{W}(C) \rfloor \{\phi \wedge (\forall \boldsymbol{x}.\, \phi_1 \rightarrow \psi_1)\}$. □

The system is obviously complete in the same sense as Hsa, since it extends it. But unlike Hsa it is also adaptation-complete.

Proposition 10 (Adaptation completeness of Hsa^{+}).
*Let $C \in$ **AComm**$^{\mathrm{SA}}$ and $\phi, \phi', \psi, \psi' \in$ **Assert** such that $\phi \# C$, (ϕ, ψ) is satisfiable, and $\models \{\phi\} \lfloor \mathcal{W}(C) \rfloor \{\psi\}$ implies $\models \{\phi'\} \lfloor \mathcal{W}(C) \rfloor \{\psi'\}$.*

*If $\vdash_{\mathsf{Hsa}^+} \{\phi\}\, C\, \{\phi \wedge \gamma\}$ for some $\gamma \in$ **Assert** such that $\models \phi \wedge \gamma \rightarrow \psi$, then $\{\phi'\}\, C\, \{\phi' \wedge (\forall \boldsymbol{x}.\, \phi \rightarrow \gamma)\}$ with $\boldsymbol{x} = \mathsf{FV}(\phi) \backslash (\mathsf{FV}(\phi') \cup \mathsf{Vars}(C))$ can be derived from that triple in system Hsa^{+}, and $\models \phi' \wedge (\forall \boldsymbol{x}.\, \phi \rightarrow \gamma) \rightarrow \psi'$.*

Proof. From $\vdash_{\mathsf{Hsa}^+} \{\phi\}\, C\, \{\phi \wedge \gamma\}$ we can apply the (conseq$_a$) rule to produce $\vdash_{\mathsf{Hsa}^+} \{\phi'\}\, C\, \{\phi' \wedge (\forall \boldsymbol{x}.\, \phi \rightarrow \gamma)\}$ with $\boldsymbol{x} = \mathsf{FV}(\phi) \backslash (\mathsf{FV}(\phi') \cup \mathsf{Vars}(C))$, since $\phi \# C$. So it just remains to prove the validity of the formula $\phi' \wedge (\forall \boldsymbol{x}.\, \phi \rightarrow \gamma) \rightarrow \psi'$.

From $\models \phi \wedge \gamma \rightarrow \psi$ it follows that $\models (\phi \rightarrow \gamma) \rightarrow (\phi \rightarrow \psi)$, and so we also have $\models (\forall \boldsymbol{x}.\, \phi \rightarrow \gamma) \rightarrow (\forall \boldsymbol{x}.\, \phi \rightarrow \psi)$. Consequently $\models \phi' \wedge (\forall \boldsymbol{x}.\, \phi \rightarrow \gamma) \rightarrow \phi' \wedge (\forall \boldsymbol{x}.\, \phi \rightarrow \psi)$, and thus $\phi' \wedge (\forall \boldsymbol{x}.\, \phi \rightarrow \gamma) \models \phi' \wedge (\forall \boldsymbol{x}.\, \phi \rightarrow \psi)$. On the other hand, as $\boldsymbol{x} \cap \mathsf{FV}(\phi') = \emptyset$, we have $\phi' \wedge (\forall \boldsymbol{x}.\, \phi \rightarrow \psi) \models \phi' \wedge (\phi \rightarrow \psi)$. Now, since $\models \{\phi\} \lfloor \mathcal{W}(C) \rfloor \{\psi\}$ implies $\models \{\phi'\} \lfloor \mathcal{W}(C) \rfloor \{\psi'\}$, it follows by Lemma 5 that $\models \phi' \rightarrow (\phi \rightarrow \psi) \rightarrow \psi'$, and hence we get $\phi' \wedge (\forall \boldsymbol{x}.\, \phi \rightarrow \gamma) \models \psi'$. Now we can conclude that $\models \phi' \wedge (\forall \boldsymbol{x}.\, \phi \rightarrow \gamma) \rightarrow \psi'$. □

Consider again the example introduced at the end of Section 2. Let K be a positive constant; the Hoare triple $\{n = K\}\, \mathsf{Fact}^{\mathsf{sa}}\, \{f_{a0} = K!\}$ can now be derived from $\{n \geq 0 \wedge n_{aux} = n\}\, \mathsf{Fact}^{\mathsf{sa}}\, \{n \geq 0 \wedge n_{aux} = n \wedge f_{a0} = n_{aux}!\}$:

$$\frac{\{n \geq 0 \wedge n_{aux} = n\}\, \mathsf{Fact}^{\mathsf{sa}}\, \{n \geq 0 \wedge n_{aux} = n \wedge f_{a0} = n_{aux}!\}}{\{n = K\}\, \mathsf{Fact}^{\mathsf{sa}}\, \{n = K \wedge (\forall n_{aux}.\, n \geq 0 \wedge n_{aux} = n \rightarrow f_{a0} = n_{aux}!)\}}$$

since $\models n = K \wedge (\forall n_{aux}.\, n \geq 0 \wedge n_{aux} = n \rightarrow f_{a0} = n_{aux}!) \rightarrow f_{a0} = K!$. As to the example at the beginning of the present section, consider the following derivation (recall that x is not assigned in P):

$$\frac{\{x > 0\}\, P\, \{x > 0 \wedge y = x\}}{\{x = K\}\, P\, \{x = K \wedge (x > 0 \rightarrow y = x)\}}$$

This proves $\{x = K\}\, P\, \{y = K\}$ since $\models x = K \wedge (x > 0 \rightarrow y = x) \rightarrow y = K$.

7 Related Work

The original notion of Static Single-Assignment (SSA) form [11] limits the syntactic occurrence of each variable as L-value of a single assignment instruction.

A construct called "Φ-function" is used to synchronize versions of the same variable used in different paths. For instance the fragment **if** $x > 0$ **then** $x := x + 10$ **else** $x := x + 20$ could be translated as $\{$**if** $x_0 > 0$ **then** $x_1 := x_0 + 10$ **else** $x_2 := x_0 + 20\}\,;\, x_3 := \Phi(x_1, x_2)$. This means that the value assigned to x_3 depends on whether execution has reached this point through the first or the second branch of the conditional. In dynamic single-assignment form [27] variables may occur in multiple assignments in different paths.

Abstracting from the fact that assignments are replaced by assume statements, the original notion of *passive form* of [14] can be seen as a kind of dynamic SA where assignment instructions are replaced by *assume* commands, but similarly to the static notion, variable synchronization is achieved by introducing fresh variables assigned in both branches. Adapting this to standard imperative syntax, the above fragment would be translated as **if** $x_0 > 0$ **then** $x_1 := x_0 + 10\,;\, x_3 := x_1$ **else** $x_2 := x_0 + 20\,;\, x_3 := x_2$. Our translation resembles the *passify* function introduced in [14], but there are significant differences: *passify* generates fresh variables abstractly, whereas we provide a concrete mechanism for this purpose; while *passify* only handles loop-free programs, our translation considers programs with loops annotated with invariants; *passify* is proved to be sound in the sense that it preserves the weakest precondition interpretation of programs, while our translation is proved to be sound with respect to the validity of Hoare triples, and moreover it is shown to be complete since it preserves derivability guided by the invariants. This is a crucial issue from the point of view of the completeness of using an intermediate SA form for verification.

Finally, *passify* does not generate version-optimal programs; the notion of *version-optimal* passive form, which uses the minimum number of version variables, is defined for unstructured programs in [16], together with a translation algorithm. In this form the above fragment becomes simply **if** $x_0 > 0$ **then** $x_1 := x_0 + 10$ **else** $x_1 := x_0 + 20$. The algorithm differs from the translation of Appendix A in that it does not contemplate annotated loops, and no proof of soundness is given. However, they are similar in the use of variables: for loop-free programs, the DSA form produced by our translation is version-optimal.

Single-assignment forms have played an important role in two different families of efficient program verification techniques.

(I) In the generation of VCs using weakest precondition computations for programs based on Dijkstra's guarded commands [12]. This setting is used as the basis for verification condition generation in ESC/Java and Boogie. For DSA programs, which appear here disguised as passive programs, weakest preconditions can be computed with quadratic size [14,23]. Note that VCs are here created by *assert* commands instead of loop conditions. The approach was extended to programs with unstructured control flow [5] in an optimized way that produces linear-size VCs. It has also been shown that efficiently provable verification conditions can be generated using instead strongest postcondition computations [16].

There exists a single semantics of guarded commands programs, given by the definition of the predicate transformers, from which a VCGen is directly derived. This stands in contrast to our approach: soundness and relative completeness of the logic and VCGen are established with respect to a standard operational semantics of *While* programs. A second important difference is the treatment of iteration. The fixpoint definition of predicate transformers for iterating commands are of no use in the verification of programs annotated with loop invariants. The approach used in ESC/Java and Boogie has been instead to convert each program into an iteration-free program, such that the verification conditions generated for the latter guarantee the soundness of the initial program (in the approach implemented in Boogie [5] loops are first converted to unstructured code with *goto* statements, but back edges are them eliminated to produce an iteration-free program). As an example consider the program shown below on the left annotated with a pre- and postcondition, and a loop invariant.

```
@requires $100 \leq x$
@ensures $x = 0$
while $(0 < x)$ {
  @invariant $0 \leq x$
  $x$ := $x - 1$
}
```

```
// assume precondition
assume $100 \leq x_0$ ;
// check invariant initialization
assert $0 \leq x_0$ ;
// assume invariant and loop condition
assume $0 \leq x_1 \wedge 0 < x_1$ ;
assume $x_2 = x_1 - 1$ ;
// check invariant preservation
assert $0 \leq x_2$ ;
// assume invariant and negated condition
assume $0 \leq x_3 \wedge \neg(0 < x_3)$ ;
// assert postcondition
assert $x_3 = 0$ ;
```

In simplified terms, this could be converted to the program shown on the right side of the figure. Observe how the fresh version variables x_1 and x_3 isolate the three relevant parts of the program (before, inside, and after the loop).

Our work differs from this in that loops are part of the intermediate SA language; they are translated into this language together with their invariants; and soundness and completeness properties are established based on the standard semantics of iteration. Annotated programs are treated explicitly, and the notion of correctly-annotated program is introduced, which allows us to distinguish between incorrect programs and correct programs containing 'wrong' invariants.

(II) In *software model checking*, where the verification is performed on a model of the system containing only bounded executions (as in *bounded model checking of software* [8]), or on an abstract model containing spurious errors (false positives, as in *predicate abstraction* [19]). In these techniques a model is usually extracted from the code by first converting it to SA form [21].

The interest and power of bounded model checking has been decidedly demonstrated in practice by the success of the CBMC tool [8]. The idea of this technique is that loops are unfolded a given number of times, and the resulting branching code is converted to a static single-assignment form (using C *conditional expressions* to encode Φ functions). Loops are thus not converted to SA form: they are eliminated by a bounded expansion before conversion. A number of transformations are then performed on the SA form, and the resulting program is easily encoded as a satisfiability problem. The transformations avoid exponential explosion, although they are not based on the observations that led to the definition of efficient predicate transformers. To the best of our knowledge, no proofs of soundness or completeness are available for bounded model checking of software.

8 Conclusion

Based on a Hoare-style logic for single-assignment programs, we have formalized a program verification technique that consists in translating annotated programs and specifications into an intermediate SA language, and generating VCs from these intermediate programs. An adaptation-complete variant of the logic is obtained by adding a dedicated consequence rule with a simple condition.

We have also shown how compact verification conditions can be produced directly from annotated SA programs. Assuming a right-associative representation of command sequences, the resulting VCs have linear size without requiring conversion of programs to unstructured form with *goto* commands as in [5].

Single-assignment intermediate forms are used extensively in software verification, both in model checking and in deductive verification; tools from both of these families eliminate iterating constructs before programs are converted to SA form. This stands in contrast with our work in this paper: we define rigorously a notion of single-assignment iterating program, and use it as the basis for a sound and complete verification technique, which includes the translation of annotated programs to SA form. We remark that the translation of loop invariants is a crucial component of the workflow, that *doesn't trivially lead to completeness.* To the best of our knowledge, this is the first time that completeness is established for a verification technique based on the use of an intermediate SA form for programs annotated with invariants.

Tools based on predicate transformers and bounded model checking incorporate many advanced features that our framework does not cover. For instance, Boogie includes automatic inference of loop invariants based on abstract interpretation, and CBMC, which natively uses a SAT (rather than SMT) solver, incorporates constant propagation and simplification functionality that is essential for making bounded verification work in practice. Still, our work here proposes a common theoretical foundation for program verification based on intermediate single-assignment form, unifying ideas from predicate transformers and program logic, while at the same time presenting adaptation-completeness as a natural property of the single-assignment setting.

In fact, a bounded notion of program verification as implemented in bounded model checking, which also relies on conversion to SA form, may also fit the same common foundation. The idea, which we will explore in future work, is to formalize this notion in a deductive setting, by obtaining a semantically justified bounded version of the VCGen of Section 4.

Acknowledgments. This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme, and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project "POCI-01-0145-FEDER-006961". The first author is also sponsored by FCT grant SFRH/BD/52236/2013.

A A Translation to SA Form

We define a translation function that transforms an annotated program into SA form. We start by introducing some auxiliary definitions to deal with variable versions. Without loss of generality, we will assume that the universe of variables of the SA programs consists of two parts: the *variable identifier* and a *version* (a non-empty list of positive numbers). We let $\mathbf{Var}^{\mathrm{SA}} = \mathbf{Var} \times \mathbb{N}^{+}$ be the set of SA variables, and we will write x_l to denote $(x, l) \in \mathbf{Var}^{\mathrm{SA}}$. We write $\Sigma^{\mathrm{SA}} = \mathbf{Var}^{\mathrm{SA}} \to D$ for the set of states, with D being the interpretation domain.

Consider the *version function* $\mathcal{V} : \mathbf{Var} \to \mathbb{N}^{+}$. The function $\widehat{\mathcal{V}} : \mathbf{Var} \to \mathbf{Var}^{\mathrm{SA}}$ is such that $\widehat{\mathcal{V}}(x) = x_{\mathcal{V}(x)}$. $\widehat{\mathcal{V}}$ is lifted to **Exp** and **Assert** in the obvious way, renaming the variables according to $\mathcal{V}$. Let $s \in \Sigma$ and $\mathcal{V} : \mathbf{Var} \to \mathbb{N}^{+}$. We define $\mathcal{V}(s) \in \mathbf{Var}^{\mathrm{SA}} \rightharpoonup D$ as the partial function $[\widehat{\mathcal{V}}(x) \mapsto s(x) \mid x \in \mathbf{Var}]$. Moreover, for $s' \in \Sigma^{\mathrm{SA}}$, $s' \oplus \mathcal{V}(s)$ denotes the overriding of s' by $\mathcal{V}(s)$.

The translation function T_{sa} is presented in Figure 7. At the bottom we show the function that receives a triple and transforms it into the SA form. This function uses the auxiliary function shown on top (with the same name but different type) that receives the initial version of each variable identifier and the annotated program, and returns a pair with the final version of each variable identifier and the SA translated program. The definition of the latter function relies in turn on various auxiliary functions that deal with the version list and version functions, and also generate renaming commands. The functions are defined using Haskell-like syntax; we assume that the renaming sequences $\mathcal{I}$ and $\mathcal{U}$, defined in the case of while commands, follow some predefined order established over **Var** (any order will do).

We will now show that T_{sa} is indeed an SA translation. The full details of the proofs and a translation example can be found in [24]. Firstly, we prove that the T_{sa} translation preserves the operational semantics of the original programs. Let us first consider some lemmas.

Lemma 9. *Let* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^{+}$*,* $s \in \Sigma$ *and* $s' \in \Sigma^{\mathrm{SA}}$*. If* $\forall x \in \mathbf{Var}.\, s(x) = s'(\widehat{\mathcal{V}}(x))$*, then* $s' = s'_0 \oplus \mathcal{V}(s)$ *for some* $s'_0 \in \Sigma^{\mathrm{SA}}$*.*

Lemma 10. *Let* $e \in \mathbf{Exp}$*,* $\phi \in \mathbf{Assert}$*,* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^{+}$*,* $s \in \Sigma$ *and* $s' \in \Sigma^{\mathrm{SA}}$*.*

1. $[\![\widehat{\mathcal{V}}(e)]\!](s' \oplus \mathcal{V}(s)) = [\![e]\!](s)$
2. $[\![\widehat{\mathcal{V}}(\phi)]\!](s' \oplus \mathcal{V}(s)) = [\![\phi]\!](s)$

Lemma 11. *Let* $C \in \mathbf{AComm}$ *and* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^{+}$*. If* $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathcal{V}', C')$*, then for every* $x \notin \mathsf{Asgn}(C)$*,* $\mathcal{V}(x) = \mathcal{V}'(x)$*.*

$$
\begin{aligned}
T_{\mathsf{sa}} &: (\mathbf{Var} \to \mathbb{N}^+) \times \mathbf{AComm} \to (\mathbf{Var} \to \mathbb{N}^+) \times \mathbf{AComm}^{\text{SA}} \\
T_{\mathsf{sa}}(\mathcal{V}, \mathbf{skip}) &= (\mathcal{V}, \mathbf{skip}) \\
T_{\mathsf{sa}}(\mathcal{V}, x := e) &= (\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))], x_{\mathsf{next}(\mathcal{V}(x))} := \widehat{\mathcal{V}}(e)) \\
T_{\mathsf{sa}}(\mathcal{V}, C_1; C_2) &= (\mathcal{V}'', C'_1; C'_2) \\
&\quad \text{where } (\mathcal{V}', C'_1) = T_{\mathsf{sa}}(\mathcal{V}, C_1) \\
&\quad \phantom{\text{where }} (\mathcal{V}'', C'_2) = T_{\mathsf{sa}}(\mathcal{V}', C_2) \\
T_{\mathsf{sa}}(\mathcal{V}, \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f) &= (\mathsf{sup}(\mathcal{V}', \mathcal{V}''), \mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then}\ C'_t; \mathsf{merge}(\mathcal{V}', \mathcal{V}'')\ \mathbf{else}\ C'_f; \mathsf{merge}(\mathcal{V}'', \mathcal{V}')) \\
&\quad \text{where } (\mathcal{V}', C'_t) = T_{\mathsf{sa}}(\mathcal{V}, C_t) \\
&\quad \phantom{\text{where }} (\mathcal{V}'', C'_f) = T_{\mathsf{sa}}(\mathcal{V}, C_f) \\
T_{\mathsf{sa}}(\mathcal{V}, \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C) &= (\mathcal{V}''', \mathbf{for}\ (\mathcal{I}, \widehat{\mathcal{V}'}(b), \mathcal{U})\ \mathbf{do}\ \{\widehat{\mathcal{V}'}(\theta)\}\ \{C'\}; \mathsf{upd}(\mathsf{dom}(\mathcal{U}))) \\
&\quad \text{where } \mathcal{I} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}(x)} \mid x \in \mathsf{Asgn}(C)] \\
&\quad \phantom{\text{where }} \mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C)] \\
&\quad \phantom{\text{where }} (\mathcal{V}'', C') = T_{\mathsf{sa}}(\mathcal{V}', C) \\
&\quad \phantom{\text{where }} \mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C)] \\
&\quad \phantom{\text{where }} \mathcal{V}''' = \mathcal{V}''[x \mapsto \mathsf{jump}(l) \mid x_l \in \mathsf{dom}(\mathcal{U})]
\end{aligned}
$$

$$
\begin{array}{ll}
\mathsf{next} : \mathbb{N}^+ \to \mathbb{N}^+ & \mathsf{sup} : (\mathbf{Var} \to \mathbb{N}^+)^2 \to (\mathbf{Var} \to \mathbb{N}^+) \\
\mathsf{next}\ (h : t) = (h + 1) : t & \mathsf{sup}\ (\mathcal{V}, \mathcal{V}')(x) = \begin{cases} \mathcal{V}(x) & \text{if } \mathcal{V}'(x) \prec \mathcal{V}(x) \\ \mathcal{V}'(x) & \text{otherwise} \end{cases} \\
\mathsf{new} : \mathbb{N}^+ \to \mathbb{N}^+ & \mathsf{merge} : (\mathbf{Var} \to \mathbb{N}^+)^2 \to \mathbf{Rnm} \\
\mathsf{new}\ l = 1 : l & \mathsf{merge}\ (\mathcal{V}, \mathcal{V}') = [x_{\mathcal{V}'(x)} := x_{\mathcal{V}(x)} \mid x \in \mathbf{Var} \wedge \mathcal{V}(x) \prec \mathcal{V}'(x)] \\
\mathsf{jump} : \mathbb{N}^+ \to \mathbb{N}^+ & \mathsf{upd} : \mathcal{P}(\mathbf{Var}^{\text{SA}}) \to \mathbf{Rnm} \\
\mathsf{jump}\ (i : j : t) = (j + 1) : t & \mathsf{upd}\ (X) = [x_{\mathsf{jump}(l)} := x_l \mid x_l \in X] \\
 & (h : t) \prec (h' : t') = h < h'
\end{array}
$$

$$
\begin{aligned}
&T_{\mathsf{sa}} : \mathbf{Assert} \times \mathbf{AComm} \times \mathbf{Assert} \to \mathbf{Assert}^{\text{SA}} \times \mathbf{AComm}^{\text{SA}} \times \mathbf{Assert}^{\text{SA}} \\
&T_{\mathsf{sa}}(\phi, C, \psi) = (\widehat{\mathcal{V}}(\phi), C', \widehat{\mathcal{V}'}(\psi)), \quad \text{where } (\mathcal{V}', C') = T_{\mathsf{sa}}(\mathcal{V}, C)\ , \text{ for some } \mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+
\end{aligned}
$$

Fig. 7: SA translation function

Lemma 12. *Let $C_t \in \mathbf{AComm}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $s_i, s_f \in \Sigma$, $s', s'_f \in \Sigma^{\text{SA}}$, $\mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C_t)]$, $T_{\mathsf{sa}}(\mathcal{V}', C_t) = (\mathcal{V}'', C'_t)$, $\mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C_t)]$. If $\langle \mathbf{while}\ b\ \mathbf{do}\ \lfloor C_t \rfloor, s_i \rangle \rightsquigarrow s_f$ and $\langle \mathbf{while}\ \mathcal{V}'(b)\ \mathbf{do}\ \{\lfloor \mathcal{W}(C'_t) \rfloor; \mathcal{U}\}; \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_f$, then $\forall x \in \mathbf{Var}.\, s_f(x) = s'_f(\widehat{\mathcal{V}'}(x))$.*

Proposition 11. *Let $C \in \mathbf{AComm}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $s_i, s_f \in \Sigma$, $s', s'_f \in \Sigma^{\text{SA}}$ and $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathcal{V}', C')$. If $\langle \lfloor C \rfloor, s_i \rangle \rightsquigarrow s_f$ and $\langle \lfloor \mathcal{W}(C') \rfloor, s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_f$, then $\forall x \in \mathbf{Var}.\, s_f(x) = s'_f(\widehat{\mathcal{V}'}(x))$.*

Proof. By induction on the structure of C. □

Secondly, we prove that lifting the translation function to Hoare triples is sound, i.e., if the translated triple is valid then the original triple is also valid.

Lemma 13. *Let* $C \in \mathbf{AComm}$*,* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$*, and* $s_i, s_f \in \Sigma$*. If* $\langle \lfloor C \rfloor, s_i \rangle \rightsquigarrow s_f$ *and* $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathcal{V}', C')$*, then* $\langle \lfloor \mathcal{W}(C') \rfloor, s'_1 \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_2 \oplus \mathcal{V}'(s_f)$*, for some* $s'_1, s'_2 \in \Sigma^{\mathrm{SA}}$*.*

Proof. By induction on the structure of C using Proposition 11. □

Proposition 12. *Let* $C \in \mathbf{AComm}$*,* $\phi, \psi \in \mathbf{Assert}$*,* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$ *and* $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathcal{V}', C')$*.*

If $\models \{\widehat{\mathcal{V}}(\phi)\}\, \lfloor \mathcal{W}(C') \rfloor\, \{\widehat{\mathcal{V}'}(\psi)\}$*, then* $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$*.*

Proof. Follows from Lemmas 13 and 10 and Propositon 11. □

Finally, we will show that Hg-derivability is preserved, i.e. if a Hoare triple for an annotated program is derivable in Hg, then the translated triple is also derivable in Hg. Again we start by stating some lemmas.

Lemma 14. *Let* $\mathcal{V}, \mathcal{V}' \in \mathbf{Var} \to \mathbb{N}^+$ *and* $\psi \in \mathbf{Assert}$*. The following hold:*

1. $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\psi)\}\, \mathsf{merge}(\mathcal{V}, \mathcal{V}')\, \{\widehat{\mathsf{sup}(\mathcal{V}, \mathcal{V}')}(\psi)\}$
2. $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\psi)\}\, \mathsf{merge}(\mathcal{V}', \mathcal{V})\, \{\widehat{\mathsf{sup}(\mathcal{V}, \mathcal{V}')}(\psi)\}$

Lemma 15. *Let* $\mathcal{I} \in \mathbf{Rnm}$ *and* $\phi \in \mathbf{Assert}$*. Then* $\vdash_{\mathsf{Hg}} \{\phi\}\, \mathcal{I}\, \{\mathcal{I}^{-1}(\phi)\}$ *holds.*

Lemma 16. *Let* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$*,* $C \in \mathbf{AComm}$*,* $\mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C)]$*,* $T_{\mathsf{sa}}(\mathcal{V}', C) = (\mathcal{V}'', C')$ *and* $\mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C)]$*. The derivation* $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}''}(\theta)\}\, \mathcal{U}\, \{\widehat{\mathcal{V}'}(\theta)\}$ *holds.*

Lemma 17. *Let* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$*,* $C \in \mathbf{AComm}$*,* $\mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C)]$*,* $T_{\mathsf{sa}}(\mathcal{V}', C) = (\mathcal{V}'', C')$*,* $\mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C)]$ *and* $\mathcal{V}''' = \mathcal{V}''[x \mapsto \mathsf{jump}(l) \mid x_l \in \mathsf{dom}(\mathcal{U})]$*. The following derivation holds* $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\psi)\}\, \mathsf{upd}(\mathsf{dom}(\mathcal{U}))\, \{\widehat{\mathcal{V}'''}(\psi)\}$ *.*

Proposition 13. *Let* $C \in \mathbf{AComm}$*,* $\phi, \psi \in \mathbf{Assert}$*,* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$ *and* $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathcal{V}', C')$*. If* $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$*, then* $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathcal{W}(C')\, \{\widehat{\mathcal{V}'}(\psi)\}$*.*

Proof. By induction on the structure of $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$. □

It is now immediate that T_{sa} conforms to Definition 6.

Proposition 14. *The* T_{sa} *function of Figure 7 is an SA translation.*

Proof. Follows directly from Propositions 12 and 13. □

References

1. Pierre America and Frank de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84(2):129–162, 1990.
2. Krzysztof R. Apt. Ten years of Hoare's logic: A survey – part 1. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.
3. John Barnes. *High integrity software: The SPARK approach to safety and security.* Addison-Wesley, 2003.
4. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.
5. Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *ACM SIGSOFT Software Engineering Notes*, 31(1):82–87, 2006.
6. Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceedings of CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.
7. Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language.* CEA LIST and INRIA, 2010.
8. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of TACAS 2004*, pages 168–176. Springer, 2004.
9. Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.
10. Loc Correnson, Pascal Cuoq, Armand Puccetti, and Julien Signoles. Frama-C user manual. Available from the Frama-C website, http://frama-c.com, 2010.
11. Ron Cytron, Jeanne Ferrante, BK Rosen, Mark N Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
12. Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics.* Springer, 1990.
13. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *Proceedings of ESOP 2013*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.
14. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proceedings of POPL 2001*, pages 193–205. ACM 2001.
15. Mike Gordon and Hélène Collavizza. Forward with Hoare. In *Reflections on the Work of C.A.R. Hoare*, History of Computing, pages 101–121. Springer, 2010.
16. Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest postcondition of unstructured programs. In *Proceedings of FTfJP 2009*, pages 6:1–6:7. ACM, 2009.
17. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
18. C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Proceeedings of Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics.* Springer, 1971.
19. Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21, October 2009.

20. Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.
21. Daniel Kroening. Software verification. In *Handbook of Satisfiability*, pages 505–532. IOS Press, 2009.
22. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In *Proceedings of ECOOP 1999*, pages 110–111. Springer, 1999.
23. K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.
24. Cláudio B. Lourenço, Maria J. Frade and Jorge S. Pinto. A Single-Assignment Translation for Annotated Programs. *ArXiv e-prints, available from http://arxiv.org/abs/1601.00584*, 2016.
25. Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *The Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.
26. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74. IEEE Computer Society, 2002.
27. Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, and Francky Catthoor. A practical dynamic single assignment transformation. *ACM Transactions on Design Automation of Electronic Systems*, 12(4):40, 2007.