Verification Conditions for Source-level Imperative Programs

Maria João Frade and Jorge Sousa Pinto

{mjf,jsp}@di.uminho.pt

Techn. Report DI-CCTC-08-01

2008, July

Computer Science and Technology Center
Departamento de Informática da Universidade do Minho
Campus de Gualtar – Braga – Portugal
http://cctc.di.uminho.pt/

DI-CCTC-08-01
Verification Conditions for Source-level Imperative Programs
by Maria João Frade and Jorge Sousa Pinto

Abstract

This paper is a systematic study of verification conditions and their use in the context of program verification. We take Hoare logic as a starting point and study in detail how a verification conditions generator can be obtained from it. The notion of *program annotation* is essential in this process. Weakest preconditions and the use of *updates* are also studied as alternative approaches to verification conditions. Our study is carried on in the context of a While language. Important extensions to this language are considered toward the end of the paper. We also briefly survey modern program verification tools and their approaches to the generation of verification conditions.

Keywords: Hoare logic, Verification Conditions, Program Verification, Program Annotations, Weakest Preconditions, Updates

1 Introduction

The idea of specifying the behaviour of programs through the use of preconditions and postconditions, and in general of assertions that are true or false relative to the current state of execution of a program, has been around since the sixties and given rise to the development of the *axiomatic* style of program semantics. The same idea has been used since the seventies in the implementation of practical tools for checking the correct behaviour of programs (vis-à-vis their specifications), marking the birth of Program Verification as a research area.

In the development of early tools [29, 21] it became clear that the most convenient way to organise a program verification system is to use the axiomatic semantics to generate first-order proof obligations (baptised *Verification Conditions*, VCs for short) that can be handled by a standard proof tool. The idea is that if all proof obligations generated for a program can be discharged (i.e. they can be proved valid), then the program is guaranteed to be correct. The semantics-based component that generates the proof obligations is called a *Verification Conditions Generator* (VCGen for short).

Program verification has recently received renewed attention from the Software Engineering community. One very general reason for this is the continuing and increasing pressure on industry to deliver software that can be certified as safe and correct. A more specific reason is that program verification methods fit very naturally the so-called Design by Contract methodology for software development, with the advent of program annotation languages like JML. Program verification fits in as the static, formal component of a methodology that encompasses also other validation methods like dynamic checking and testing.

But recent times have not only witnessed an increase in the popularity of program verification with software engineers; the field itself has developed and exciting new techniques were recently proposed to target the certification requirements that have been created, for instance, by mobile code and the associated new architectures for the execution of software. For instance the following two techniques both rely on verification conditions.

- *Proof-carrying code* [44], based on the generation of verification conditions from annotated low-level (compiled) code. The idea is that a compiler can automatically produce a proof – a certificate – that the compiled code satisfies some requirements (say, it performs only safe memory accesses), and an execution platform may then (cheaply, and without relying on the behaviour of complex pieces of software like compilers or theorem provers) generate VCs and check that the certificate provides evidence for these VCs.
- Certain information-flow properties (such as non-interference) that have traditionally been treated using a number of language-based Security techniques [48] may also be addressed using axiomatic semantics tools like Hoare logic [7] or Dynamic logic [15], providing yet further motivation for the relevance of Logic-based methods for reasoning about programs.

This paper is a study of Verification Conditions for imperative, sequential, high-level programming languages. Our tutorial presentation treats annotated programs formally, and provides a uniform development of VCGen algorithms from program logics. Although we do not attempt to give a thorough survey of program logics or program verification systems, we do examine (and provide references for) those systems that have become more popular in recent years.

The paper is structured as follows. Section 2 sets the basis by introducing a simple programming language and the notion of Hoare triple. In Section 3 the inference system H of Hoare logic is introduced, and Section 4 discusses its use in program verification, based on the generation of Verification Conditions. Section 5 presents an alternative, goal-directed formulation of Hoare logic (system Hg), that is more amenable to mechanising the construction of proof trees, since it contains no ambiguity in the choice of rule. In Section 6 we show how the introduction of program annotations (resulting in system Hga) completes this progression toward the mechanisation of Hoare logic.

The previous approach forces the insertion of annotations between any two composed commands of a program. The next step is to eliminate the need for this tedious process. This can be done in two ways: by backward propagation of assertions, and by forward propagation. The former approach leads us in Section 7 to VCGens based on Dijkstra's weakest preconditions. The latter approach, covered in Section 9, takes us in the direction of systems based on the use of *updates*. Section 10 discusses how the simple language used here can be extended with a number of features that can be found in realistic programming languages, and Section 11 concludes the paper.

We also briefly review verification condition generation tools that bring to practice these ideas. Section 8 presents a guarded command language in the style introduced by Dijkstra, and discusses how it has been used as an intermediate language in the VCGens of the ESC family of tools. The KeY tool, based on a variant of Dynamic logic, is discussed in Section 9; the Boogie and Why tools, both generic VCGens, are discussed in Section 11. Our goal here is not to exhaustively cover all tools for program verification; we merely select a few that we find suitable to illustrate the concepts discussed in the paper.

2 Programs and Specifications: Hoare Triples

In this paper we explore methods for specifying programs in a simple imperative language and for proving the correctness of such specifications formally. We consider only *partial correctness* specifications: programs are required to behave properly if they terminate, but are not required to terminate.

We consider a typical While language with data types for integer numbers and booleans. Commands include a do-nothing command, assignment, composition, while loop and (two branched) conditional execution. The language has two base types

$$\tau ::= \textbf{bool} \mid \textbf{int}$$

The syntax of boolean and integer expressions could be more or less evolved; boolean expressions should contain the boolean constants `true` and `false`, and it must be possible to form expressions by comparing the values of two integers. The language also includes boolean operators for conjunction, disjunction, and negation (we use C/Java-like syntax for operators). Integer expressions are formed from constants and a set of variables $\mathcal{V}$, together with a number of operators on integers. We let $x, y, ...$ range over $\mathcal{V}$. Note that these choices have no impact on the material presented in the paper; the reader will find in the literature presentations that use simpler languages as well as richer ones.

The key semantic notion is that of the program *state* (given by the values of the variables involved in the computation). The value of an expression depends on the current state, and the computing device changes the state when it executes a command. In addition to expressions and commands, we need syntax for formulas that express assertions about properties of particular states, as well as a class of formulas for specifying the bahaviour of programs. We have the following *phrase types*

$$\theta ::= \textbf{Exp}[\tau] \mid \textbf{Comm} \mid \textbf{Assert} \mid \textbf{Spec}$$

corresponding respectively to *expressions* (for each data type), *commands*, *assertions* and *specifications*. Their abstract syntax is defined in Figure 1. Note that the

$$
\begin{array}{llcl}
\mathbf{Exp[int]} & \ni e & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x \mid -e \mid e+e \mid e-e \mid e*e \mid e\,\texttt{div}\,e \mid e\,\texttt{mod}\,e \\
\mathbf{Exp[bool]} & \ni b & ::= & \texttt{true} \mid \texttt{false} \mid e == e \mid e < e \mid e <= e \mid e > e \mid e >= e \mid e\,!= e \\
 & & \mid & b\ \&\&\ b \mid b \parallel b \mid !\,b \\
\mathbf{Comm} & \ni C & ::= & \mathbf{skip} \mid C\,;\,C \mid x := e \mid \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ b\ \mathbf{do}\ C \\
\mathbf{Assert} & \ni A & ::= & \texttt{true} \mid \texttt{false} \mid e == e \mid e < e \mid e <= e \mid e > e \mid e >= e \mid e\,!= e \\
 & & \mid & !\,A \mid A\ \&\&\ A \mid A \parallel A \mid A \rightarrow A \mid \texttt{Forall}\,x.\,A \mid \texttt{Exists}\,x.\,A \\
\mathbf{Spec} & \ni S & ::= & \{A\}\,C\,\{A\}
\end{array}
$$

Fig. 1. Abstract Syntax

syntax rules may also be read implicitly as typing rules. For instance, if e_1, e_2 have type **Exp**[**int**], then so has $e_1 + e_2$, and so on.

Expressions and commands are fairly obvious. It remains to discuss the language of assertions, i.e. properties that hold at a given point in the program execution, and program specifications.

It is common for assertions to be defined as a super-set of boolean expressions, since they may have to refer to the values of expressions in the current state of the program. If exactly the same syntax is used for assertions and boolean expressions, it will be easier for ordinary programmers to write specifications (and also *program annotations*, see Section 6).

We thus construct our language of assertions starting from the language of boolean expressions of the programming language, and extend that with an implication connective. Moreover, to allow for first-order reasoning about programs, universal and existential quantifiers are introduced. So basically our language of assertions is a first-order logic, where the terms and predicates are respectively integer expressions and binary comparison predicates. We do not include many common features of first-order logic (like arbitrary predicate symbols) in order to keep the presentation concise.

Again, the particular choice for the language of assertions is merely illustrative; the bottom line is that the richer the programming language is (and the richer the set of *types* it contains), the richer the assertion language (and its semantics) must be. Accordingly, the logical theories required for proving the validity of assertions will also have to be progressively richer.

Assertions that hold before and after execution of a program – *preconditions* and *postconditions* respectively – will allow one to write specifications of programs or *Hoare Triples* – the last syntactic category in Figure 1. The intuitive meaning of a specification $\{P\}\,C\,\{Q\}$ is that if the program C is executed in an initial state in which the assertion (precondition) P is *true*, then either execution of C does not terminate or, if it does, assertion Q (a postcondition) will be *true* in the final state. Because termination is not guaranteed, this notion is called a *partial correctness* specification.

The notions of specification and Hoare triple coincide. We remark however that sometimes it is useful to consider a notion of specification (P, Q) consisting only of a precondition P and a postcondition Q. In this view if the Hoare triple $\{P\}\,C\,\{Q\}$ is valid for some program C then C is said to be correct with respect to the specification (P, Q). We will use both notions in the paper but this shouldn't generate any confusion.

The introduction of binding quantifiers in assertions imposes the usual notions of free and bound variables. Variables that occur in the program C (and possibly also free in the precondition P or in the postcondition Q) are called *program variables.* Variables that occur free in P or Q but not in the program will be called *auxiliary* or *ghost variables* (their use will be explained in Section 7), and those that are bound by some quantifier in P or Q will be called *logical variables*.

Semantics. The meaning of a grammatically correct program can be formalized in different ways (see [53, 46]):

- *Operational semantics* is focused on the computation the program induces on a machine (*small-step*, or *structural*, if the emphasis is on the individual steps of the execution; *big-step*, or *natural semantics*, if the emphasis is on the relationship between the initial and the final state of the execution).
- *Denotational semantics* is focused on representing the effect of executing a program by a mathematical object.

Frequently, the logical system for proving partial correctness properties of programs is viewed as an *axiomatic semantics*, focused on specific properties (expressed by assertions) of the effect of executing a program.

In the following, a natural semantics is used to describe the meaning of commands, and we define semantic functions to interpret expressions, assertions and specifications in a denotational style.

The semantics is given in terms of states. The base types are interpreted as expected

$$\begin{aligned} [\![\mathbf{bool}]\!] &= \{\mathit{true}, \mathit{false}\} \\ [\![\mathbf{int}]\!] &= \mathbb{Z} \end{aligned}$$

Boolean and integer expressions are interpreted as boolean or integer values, but these values depend on the values of variables that may occur in the expressions. In other words, they depend on a *state*, which is a function that maps each variable into its integer value[1]. We write $\Sigma = \mathcal{V} \to [\![\mathbf{int}]\!]$ for the set of states, and for $s \in \Sigma$, $y \in \mathcal{V}$ and $v \in [\![\mathbf{int}]\!]$, $s[y \mapsto v]$ denotes the following state

$$s[y \mapsto v](x) = \begin{cases} v & \text{if } \ x = y \\ s(x) & \text{if } \ x \neq y \end{cases}$$

For each phrase type we define the corresponding *domain of interpretations* (the set of possible meanings)

$$\begin{aligned} [\![\mathbf{Exp}[\tau]]\!] &= \Sigma \to [\![\tau]\!] \\ [\![\mathbf{Comm}]\!] &\subseteq \Sigma \times \Sigma \\ [\![\mathbf{Assert}]\!] &= \Sigma \to \{\mathit{true}, \mathit{false}\} \\ [\![\mathbf{Spec}]\!] &= \{\mathit{true}, \mathit{false}\} \end{aligned}$$

These domains reflect our assumption that an expression has a value at every state (evaluation of expressions always terminates without an error stop, see below). and that expression evaluation never changes the state (the language is free of *side effects*).

The behaviour of a command is to transform the state of a computation. Commands are interpreted operationally via the *evaluation* relation $(\cdot,\cdot) \Downarrow \cdot \subseteq \mathbf{Comm} \times \Sigma \times \Sigma$. Intuitively $(C, s) \Downarrow s'$ means that the execution of C from s will terminate and the resulting state will be s'. A command whose execution does not terminate for an initial state s is absent from this interpretation since there is no pair (s, s') corresponding to it.

Denotationally, the meaning of a command would be a state-transformation function $\Sigma \to \Sigma$, and dealing with non-termination would require the introduction of more sophisticated mathematical domains. This denotational interpretation is not required (nor the most appropriate) for our present goal.

[1] Another possibility would be to consider states as partial functions.

Expressions:

$$
\begin{aligned}
[\![n]\!](s) &= n \qquad \text{for} \quad n \in \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
[\![x]\!](s) &= s(x) \\
[\![-e]\!](s) &= -[\![e]\!](s) \\
[\![e_1 \Box e_2]\!](s) &= [\![e_1]\!](s) \boxdot [\![e_2]\!](s) \quad , \\
&\quad \text{for} \quad (\Box, \boxdot) \in \{(+, +), (-, -), (*, \times), (\,\texttt{div}\,, \div), (\,\texttt{mod}\,, \texttt{mod}\,)\} \\
[\![\texttt{true}]\!](s) &= \mathit{true} \\
[\![\texttt{false}]\!](s) &= \mathit{false} \\
[\![\,!\,e]\!](s) &= \neg[\![e]\!](s) \\
[\![e_1 \Box e_2]\!](s) &= [\![e_1]\!](s) \boxdot [\![e_2]\!](s) \quad , \\
&\text{for} \quad (\Box, \boxdot) \in \{(==, =), (\,!=, \neq), (<, <), (<=, \leq), (>, >), (>=, \geq)\}
\end{aligned}
$$

Commands:

1. $(\mathbf{skip}, s) \Downarrow s$
2. $(x := e, s) \Downarrow s[x \mapsto [\![e]\!](s)]$
3. If $(C_1, s) \Downarrow s'$ and $(C_2, s') \Downarrow s''$ then $(C_1\,;\,C_2, s) \Downarrow s''$
4. If $(C_t, s) \Downarrow s'$ and $[\![b]\!](s) = \mathit{true}$ then $(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, s) \Downarrow s'$
5. If $(C_f, s) \Downarrow s'$ and $[\![b]\!](s) = \mathit{false}$ then $(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, s) \Downarrow s'$
6. If $(C, s) \Downarrow s', (\mathbf{while}\ b\ \mathbf{do}\ C, s') \Downarrow s''$, and $[\![b]\!](s) = \mathit{true}$ then $(\mathbf{while}\ b\ \mathbf{do}\ C, s) \Downarrow s''$
7. If $[\![b]\!](s) = \mathit{false}$ then $(\mathbf{while}\ b\ \mathbf{do}\ C, s) \Downarrow s$

Assertions:

$$
\begin{aligned}
[\![\texttt{true}]\!](s) &= \mathit{true} \\
[\![\texttt{false}]\!](s) &= \mathit{false} \\
[\![\,!\,A]\!](s) &= \neg[\![A]\!](s) \\
[\![A_1 \Box A_2]\!](s) &= [\![A_1]\!](s) \boxdot [\![A_2]\!](s) \quad , \text{for} \quad (\Box, \boxdot) \in \{(\,\&\&\,, \wedge), (\,\|\,, \vee), (\rightarrow, \Rightarrow)\} \\
[\![\texttt{Forall}\,x.\,A]\!](s) &= \forall v \in [\![\mathbf{int}]\!].\ [\![A]\!](s[x \mapsto v]) \quad , \text{with } v \text{ fresh} \\
[\![\texttt{Exists}\,x.\,A]\!](s) &= \exists v \in [\![\mathbf{int}]\!].\ [\![A]\!](s[x \mapsto v]) \quad , \text{with } v \text{ fresh} \\
[\![e_1 \Box e_2]\!](s) &= [\![e_1]\!](s) \boxdot [\![e_2]\!](s) \quad , \\
&\text{for} \quad (\Box, \boxdot) \in \{(==, =), (\,!=, \neq), (<, <), (<=, \leq), (>, >), (>=, \geq)\}
\end{aligned}
$$

Specifications:

$$[\![\{P\}\,C\,\{Q\}]\!] = \forall s, s' \in \Sigma.\ [\![P]\!](s) \wedge (C, s) \Downarrow s' \Rightarrow [\![Q]\!](s')$$

Fig. 2. Semantic equations and natural semantics for While

Assertions are interpreted as truth values depending on a valuation function given by the state, and specifications are interpreted as truth values independently of states.

Figure 2 shows the semantic equations that define the interpretation functions for expressions, assertions and specifications, and the natural semantics for commands as a set of evaluation rules. Note that we make an overloaded use of $[\![.]\!]$ (we could subscript the semantic brackets with the phrase type of the object phrase that is being interpreted, but this is usually obvious from the phrase itself or from the context).

Concerning the interpretation of expressions, note that we are assuming that $n \div 0$ and $n\,\mathsf{mod}\,0$ produce some erroneous (and fixed) integer result. To treat the detection of arithmetic errors it would be necessary to extend our current semantics, for instance by enlarging the domains of interpretation of integer and boolean expressions to include one or more special results denoting errors.

Note that the semantic interpretation of a command C can be seen as a partial function, since the binary relation on states induced by C satisfies the following property.

Lemma 1 (Determinacy) *If* $(C, s) \Downarrow s'$ *and* $(C, s) \Downarrow s''$, *then* $s' = s''$.

Proof. By induction on the structure of C.

The semantic interpretation of assertions is the usual for a first-order logic. Observe the correspondence between the logical connectives of the assertion language and the connectives of classic predicate calculus used in the meta-logic. Let A be an assertion and $s \in \Sigma$. If $[\![A]\!](s) = \mathit{true}$, we say that A *holds for* s. When $[\![A]\!](s) = \mathit{true}$

for all states $s \in \Sigma$, A is said to be *valid*, written $\models A$. For a set of assertions $\mathcal{M}$, we write $\models \mathcal{M}$ if $\models A$ holds for every $A \in \mathcal{M}$.

The semantic interpretation of a Hoare triple is a truth value that is independent of states. However there is a quantification over the set of states. Notice that a command C trivially satisfies a specification when the execution of C fails to terminate (since there exists no s such that $(C, s) \Downarrow s'$). A Hoare triple $\{P\}\, C\, \{Q\}$ is true if and only if, for all states satisfying P, executing C either fails to terminate or terminates in a state satisfying Q. We will also call a specification *valid* when it is true.

Total Correctness. The above notion corresponds to *partial correctness* specifications, since termination is not guaranteed. If termination is required, we are in the presence of a *total correctness* formula and write $[P]\, C\, [Q]$ instead. The idea is that $[P]\, C\, [Q]$ is true if and only if, for all states satisfying P, executing C terminates in a state satisfying Q. Note that the validity of a total correctness specification can be proved by proving the corresponding partial correctness specification and additionally proving termination of commands.

We will leave the discussion of termination out of this paper, since it is rather different in nature. Most program verification tools allow the users to specify *loop variants*: integer expressions whose value strictly decreases with each iteration [23].

3 Hoare Logic

Given a specification in the form of a Hoare triple $\{P\}\, C\, \{Q\}$, how can its validity be checked? One could think of *testing* the program by running it with a battery of initial states satisfying the precondition P, and checking whether Q is satisfied after execution if the program terminates. Of course, this process can never be exhaustive, and one can only expect to have a certain degree of confidence about the validity of a specification. This would not provide a proof of correctness.

The usual method for assuring the validity of specifications is to use a sound program-proof system. By sound we mean that it should not infer specifications that are not valid. We will now outline a formal system for inferring valid specifications and describe methods for mechanically verifying the correctness of given specifications.

A program-proof system is a set of *inference rules* that can be seen as fundamental laws about programs. Before defining an inference system for our While-language programs, let us first explain some basic concepts.

An *inference rule* consists of zero or more *premises* and a single *conclusion* (we use the notation of separating the premises from the conclusion by a horizontal line). Each rule consisting in a number of premises and a conclusion is in fact a *schema* for a specification (that is, a pattern containing meta-variables, each ranging over some phrase type). An *instance* of an inference rule is obtained by replacing all occurrences of each meta-variable by a phrase in its range. In some rules, there may be side conditions that must be satisfied by the replacement. Also, there may be syntactic operations (such as substitutions) that must be carried out after the replacement. An inference rule containing no premises is called an *axiom schema* (or simply, *axiom*). An inference rule is *sound* if and only if, for every instance, if the premises are all valid then the conclusion is valid.

Deductions will be presented as trees; nodes will be labelled with formulas (specifications); the specification being proved lies at the root node, and the immediate sub-trees are deductions corresponding to the premises of the rule instance used to infer the root specification formula. A *deduction tree* is also called a *derivation tree* or *proof tree*, and it constitutes a formal proof of the specification at its root.

$$(skip)\quad \frac{}{\{P\}\,\mathbf{skip}\,\{P\}} \qquad (assign)\quad \frac{}{\{Q[x \mapsto e]\}\; x := e\; \{Q\}}$$

$$(seq)\quad \frac{\{P\}\, C_1\, \{R\} \qquad \{R\}\, C_2\, \{Q\}}{\{P\}\, C_1 \,;\, C_2\, \{Q\}}$$

$$(while)\quad \frac{\{I \;\&\&\; b\}\; C\; \{I\}}{\{I\}\; \mathbf{while}\; b\; \mathbf{do}\; C\; \{I \;\&\&\; !\, b\}}$$

$$(if)\quad \frac{\{P \;\&\&\; b\}\; C_t\; \{Q\} \qquad \{P \;\&\&\; !\, b\}\; C_f\; \{Q\}}{\{P\}\; \mathbf{if}\; b\; \mathbf{then}\; C_t\; \mathbf{else}\; C_f\; \{Q\}}$$

$$(conseq)\quad \frac{\{P\}\; C\; \{Q\}}{\{P'\}\; C\; \{Q'\}} \quad \text{if } \models P' \rightarrow P \text{ and } \models Q \rightarrow Q'$$

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

The inference system of Hoare logic is shown in Figure 3. We name it system H. Moreover, we say that a Hoare triple $\{P\}\, C\, \{Q\}$ is *derivable* in this system, and write $\vdash_{\mathsf{H}} \{P\}\, C\, \{Q\}$, if it can be inferred from the rules of Figure 3 (in general we will subscript the symbol $\vdash$ with an inference system to denote derivability in that system).

This system comprises one rule for each command construct in the programming language, and the consequence rule, which allows us to derive a specification from another specification by strengthening the precondition or weakening the postcondition. Two of the command rules (*skip* and *assign*) are in fact axioms (since the do-nothing and assignment commands have no sub-commands, the rules have no premises).

The assignment rule states that a postcondition Q can be granted for a command $x := e$ if the condition that results from substituting e for x in Q holds as precondition. $Q[x \mapsto e]$ stands for the result of substituting e for the free occurrences of x in Q[2]. The sequencing and conditional constructs are handled by rules that are quite straightforward to understand. The rule for while commands uses the familiar notion of an *invariant* condition (denoted I in the rule), preserved by executions of the loop body. Note that one does not require that condition I holds *throughout* execution of the loop body; only that it is reestablished at the end of each iteration. We remark that, as the assertion language subsumes boolean expressions, the boolean condition b can be combined with assertions in the (if) and $(while)$ rules (otherwise an embedding function would have to be applied to b).

Finally the $(conseq)$ rule establishes a connection with predicate logic by means of side conditions that are assertions rather that specifications. The idea is that a specification can be derived from another specification provided that their corresponding preconditions and postconditions are related in the way dictated by the side conditions.

As a trivial example of using this system, consider the following proof tree concerning a single-instruction program, which is built from an instance of rule schema

[2] In the application of a substitution to a term, we rely on a variable convention. The action of a substitution over a term is defined, as usual, with possible renaming of bound variables.

(*conseq*) and an instance of axiom schema (*assign*).

$$\frac{\overline{\{x+1>10\}\, x := x+1\, \{x>10\}}}{\{x>10\}\, x := x+1\, \{x>10\}} \text{ if } \models x>10 \rightarrow x+1>10 \text{ and } \models x>10 \rightarrow x>10$$

We remark that the side conditions concern the validity of purely first-order formulas (with no occurrences of program constructs). The compositional rules, associated with the program constructs of the language, depend on the semantics of commands but are *independent of the interpretation of data types and arithmetic operations.* It is the rule of consequence that brings data-specific assertions to bear on the proofs of specifications, through the introduction of side conditions.

Different presentations of the consequence rule can be found in the literature; some authors prefer to include the side conditions as premises in the rule. Observe however that care must be taken interpreting the following very common presentation of the rule

$$\frac{P' \rightarrow P \qquad \{P\}\, C\, \{Q\} \qquad Q \rightarrow Q'}{\{P'\}\, C\, \{Q'\}}$$

In particular, program variables that occur in $P' \rightarrow P$ and $Q \rightarrow Q'$ must be seen as universally quantified *locally* to those formulas, and not globally in the rule, which would be the usual interpretation in a natural deduction-like presentation. This is due to the fact that it is the validity, rather than truth, of $P' \rightarrow P$ and $Q \rightarrow Q'$ that must be established.

Because of the presence of the consequence rule, Hoare logic is not meant to be used by itself; it must always be accompanied by some device for proving the validity of side-conditions. Since assertion languages are usually variations of first-order logic, this device will typically be a predicate calculus. Moreover, this first-order reasoning will have to use specific theories to cope with the data types that are present in the language. (for the concrete case of the assertion language considered in this paper, we assume a predicate calculus with a theory for integers).

Finally, note that it is an immediate consequence of the above discussion that reasoning about programs (with a first-order assertion language) is in general not decidable.

We will now address the relationship between system H and the semantics of the language. All of the subsequent material relies on the fact that the inference system is sound: if some Hoare triple can be proved using system H, then it is indeed valid according to the semantics.

Proposition 1 (Soundness) *In system* H*, every derivable specification is semantically valid. That is, if* $\vdash_{\mathsf{H}} \{P\}\, C\, \{Q\}$*, then* $[\![\{P\}\, C\, \{Q\}]\!] = \mathit{true}$*.*

Proof. By induction on the derivation of $\vdash_{\mathsf{H}} \{P\}\, C\, \{Q\}$. For the while case we also proceed by induction on the definition of the evaluation relation.

The classic reference on Hoare logic is [26]. Two papers survey its technical development [2] and its historical development and impact [33]. In this paper we focus on applying Hoare logic in the context of program verification; its use in the software development process is covered in detail in textbooks [4, 50].

Bertot [10] has formalized in the Calculus of Inductive Constructions the semantics (both operational and denotational) of a language like the one considered in this paper. A so-called *deep embedding* of the inference system of Hoare logic is given (the rules are encoded as cases of the inductive definition of a predicate), and correctness with respect to the operational semantics is proven. This very instructive work is available as part of an integrated Coq development that covers many aspects of the semantics of programming languages. We will come back to this development in Section 7.

4 Verification Conditions and VCGens

With the help of a theorem prover or proof assistant, Hoare logic can be put into practice to produce a program verification system. This can be done in two ways. The first is by directly encoding the inference system in the logic of the proof tool and reasoning simultaneously with rules of both Hoare logic and first-order logic as required: reasoning starts with the former but switches to the latter logic when side conditions are introduced by the consequence rule.

The alternative approach, which is prevalent in modern program verification systems, is organised in two steps as follows:

1. A proof tree is constructed for the desired specification, *assuming that the side conditions generated by the consequence rule are valid.*
2. An external proof tool is used (such as a theorem prover or a proof assistant) to actually establish the validity of the side conditions.

We let $\mathcal{A} \Vdash_{\mathsf{H}} \{P\}\, C\, \{Q\}$, where $\mathcal{A}$ represents a set of first-order assertions, denote the fact that there exists a derivation tree of $\{P\}\, C\, \{Q\}$ in system H such that $\mathcal{A}$ is the set of side conditions in that tree (in general we will subscript the symbol $\Vdash$ with an inference system to denote the existence of a proof tree in that system with a given set of side conditions).

Note that the soundness of this approach to verification is immediate: if $\mathcal{A} \Vdash_{\mathsf{H}} \{P\}\, C\, \{Q\}$ and all the assertions of $\mathcal{A}$ hold, then $\vdash_{\mathsf{H}} \{P\}\, C\, \{Q\}$. The first step is said to generate *proof obligations* (the elements of $\mathcal{A}$) that must then be *discharged* in the second step. In the context of program verification these proof obligations are usually known as *verification conditions*.

The advantage of this method lies in its flexibility. Since program constructs do not occur in the assertions in $\mathcal{A}$, the second step involves only discharging first-order proof obligations, for which a great number of proof tools can be used interchangeably or even cooperatively. It is also easier to modify a program verification system organized in this way. For instance if the programming language is modified, only the first step above is affected.

Clearly the inference system allows for different proof trees to be constructed for the same conclusion specification, and in fact these trees may well have different sets of side conditions. This tree construction process can be replaced by a simple algorithm – a *Verification Conditions Generator* (VCGen) – that constructs a set of verification conditions by applying a specific strategy, i.e. the algorithm produces verification conditions that correspond to the side conditions of one particular derivation.

VCGen algorithms will be given as functions that take as input a Hoare triple and return a set of first-order proof obligations. A side condition of the form $\models P \rightarrow Q$ will give rise to a verification condition written as $[\, P \rightarrow Q \,]$, where the $[\,\cdot\,]$ notation represents the processing that may be required to export proof obligations to the target proof tool.

This processing may require more than just translating assertions into the language of the tool; one typical operation corresponds to calculating the *universal closure* of an assertion, by making explicit the universal quantification over program variables that is implicit in the notion of validity (i.e. truth in all states) of side conditions. For instance, the side condition $\models x > 10 \rightarrow x + 1 > 10$ in the earlier example would generate the verification condition $[\, x > 10 \rightarrow x + 1 > 10 \,]$, which in turn could be exported as the formula $\texttt{Forall}\, x.\, x > 10 \rightarrow x + 1 > 10$.

We remark that there are two possible sources of errors that may cause the verification of a given Hoare triple to fail. These are:

1. *program errors*; and

2. *specification errors*: the program may be correct with respect to the intended specification, but the specification has not been correctly formalized (there are errors in the preconditions or postconditions).

In the rest of this paper we will study several VCGen algorithms and show how they are obtained from the inference system H or some other related system. Our first step is to study how the construction of derivations of H can be mechanized.

5 An Alternative Formulation of Hoare Logic

We now focus on using Hoare logic to produce verification conditions, as part of the verification architecture outlined in the previous section. Given a specification, we wish to construct a proof tree having that specification as conclusion; verification conditions result from the side conditions of instances of rules in that tree.

There are two desirable properties that the inference system of Hoare logic should enjoy to make possible the automatic construction of proof trees.

1. The *sub-formula* property: the premises of a rule should not contain occurrences of assertions that do not occur in the rule's conclusion. In other words, all the assertions that occur in the premises should be sub-formulas of those occurring in the conclusion. Otherwise one would have to invent formulas when applying the rule in a backward fashion.
2. Unambiguity: a unique rule should be applicable in a backward fashion for any given goal, so that the construction of derivation trees can be syntax-directed.

We start with the second property. The system H of Hoare logic can easily be transformed into an equivalent unambiguous system; observe that ambiguity is only caused by the presence of the consequence rule. In particular, note that some rules are only applicable to goals that satisfy certain constraints, namely:

– the (*skip*) rule can only be applied if the precondition and the postcondition are equal;
– the (*assign*) rule for assignment can only be applied if the precondition results from the postcondition by performing the corresponding substitution;
– the (*while*) rule can only be applied if the precondition is an invariant of the loop, and the postcondition is the same invariant strengthened with the negation of the loop condition.

Application of these three rules to goals with arbitrary preconditions and postconditions may previously require the application of the consequence rule. This becomes unnecessary if the weakening/strengthening conditions are judiciously distributed through the program rules. This is precisely the case in the system of Figure 4.

This new system may be called a *goal-directed* system, since it consists of exactly one rule for each program construct, and moreover, for a given specification $\{P\}\,C\,\{Q\}$, the rule matching the program C can *always* be applied (if the side conditions are met), since the precondition and postcondition are now arbitrary. We name it system Hg, and we let $\vdash_{\mathsf{Hg}} \{P\}\,C\,\{Q\}$ denote the fact that $\{P\}\,C\,\{Q\}$ can be inferred from the rules of Figure 4.

It is easy to see that the (*conseq*) rule is admissible in system Hg, and that systems H an Hg are equivalent.

Lemma 2 *If* $\vdash_{\mathsf{Hg}} \{P\}\,C\,\{Q\}$ *and both* $\models P' \rightarrow P$ *and* $\models Q \rightarrow Q'$ *hold, then* $\vdash_{\mathsf{Hg}} \{P'\}\,C\,\{Q'\}$.

Proof. By induction on the derivation of $\vdash_{\mathsf{Hg}} \{P\}\,C\,\{Q\}$.

$$\frac{}{\{P\}\,\mathbf{skip}\,\{Q\}} \text{ if } \models P \rightarrow Q \qquad \frac{}{\{P\}\,x := e\,\{Q\}} \text{ if } \models P \rightarrow Q[x \mapsto e]$$

$$\frac{\{P\}\,C_1\,\{R\} \qquad \{R\}\,C_2\,\{Q\}}{\{P\}\,C_1\,;\,C_2\,\{Q\}}$$

$$\frac{\{I \,\&\&\, b\}\,C\,\{I\}}{\{P\}\,\mathbf{while}\ b\ \mathbf{do}\ C\,\{Q\}} \text{ if } \models P \rightarrow I \text{ and } \models I \,\&\&\, !\,b \rightarrow Q$$

$$\frac{\{P \,\&\&\, b\}\,C_t\,\{Q\} \qquad \{P \,\&\&\, !\,b\}\,C_f\,\{Q\}}{\{P\}\,\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\,\{Q\}}$$

Fig. 4. Goal-directed version of Hoare logic – System Hg

Proposition 2 $\vdash_{\mathsf{H}} \{P\}\,C\,\{Q\}$ *iff* $\vdash_{\mathsf{Hg}} \{P\}\,C\,\{Q\}$.

Proof. ⇒) By induction on the derivation of $\vdash_{\mathsf{H}} \{P\}\,C\,\{Q\}$, using Lemma 2. ⇐) By induction on the derivation of $\vdash_{\mathsf{Hg}} \{P\}\,C\,\{Q\}$.

We remark that this new system still does not enjoy the sub-formula property. In fact, although we have removed two rules that did not enjoy the property, we have also caused the while rule to lose that property since the invariant assertion no longer occurs in the conclusion. Moreover the rule for the sequence construct does not enjoy the property either, since the intermediate assertion R does not occur in the conclusion.

In fact, attaining the sub-formula property, and the possibility of automated proof construction, requires the programmer to provide some extra information.

6 Program Annotations

Program verification cannot in general be carried out in a totally automatic way. The use of a first-order assertion language may require using interactive proof development instead of automatic theorem proving. The more information is provided in the source program, the higher the chances of mechanization are. Loop invariants, for instance, are typically provided manually as inputs to the verification process.

One way to restore the sub-formula property in our current system for Hoare logic is precisely to introduce human-provided *annotations* in the programs. An *annotated program* is a program with assertions embedded within it. Inserted assertions should express conditions one expects to hold whenever control reaches the points at which they occur. Let **AComm** be the class of *annotated commands*. Its abstract syntax is defined by

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

For instance the annotated command

$$\mathbf{while}\ b\ \mathbf{do}\,\{I\}\,C$$

denotes a loop with condition b, instruction body C, and (user-provided) invariant I. Naturally, the introduction of the loop invariant plays no role in the execution semantics of the loop. Similarly, the command sequence

$$C_1\,;\,\{R\}\;C_2$$

$$\frac{}{\{P\}\,\mathbf{skip}\,\{Q\}}\quad \text{if } \models P \rightarrow Q \qquad\qquad \frac{}{\{P\}\,x := e\,\{Q\}}\quad \text{if } \models P \rightarrow Q[x \mapsto e]$$

$$\frac{\{P\}\,C_1\,\{R\} \qquad \{R\}\,C_2\,\{Q\}}{\{P\}\,C_1\,;\,\{R\}\;C_2\,\{Q\}}$$

$$\frac{\{I\ \&\&\ b\}\,C\,\{I\}}{\{P\}\,\mathbf{while}\ b\ \mathbf{do}\,\{I\}\,C\,\{Q\}}\quad \text{if } \models P \rightarrow I \ \text{ and } \models I\ \&\&\ !\,b \rightarrow Q$$

$$\frac{\{P\ \&\&\ b\}\,C_t\,\{Q\} \qquad \{P\ \&\&\ !\,b\}\,C_f\,\{Q\}}{\{P\}\,\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\,\{Q\}}$$

Fig. 5. Goal-directed version of Hoare logic for annotated programs – System Hga.

has the same semantics as $C_1\,;\,C_2$, but is annotated with an assertion that must be true when the execution of C_1 (started in a state in which any given preconditions of the combined command hold) terminates.

An inference system of Hoare logic for annotated programs is shown in Figure 5. We name it system Hga. Note that this system enjoys the sub-formula property, and can be used mechanically in a backward fashion to generate verification conditions. We will exemplify its use with a classic program. This example also illustrates the need to extend the assertion language with a vocabulary of function symbols.

a) Program C_{Fib}

```
x := 1 ;
y := 0 ;
i := 1 ;
while i < n do
{
    aux := y ;
    y := x ;
    x := x + aux ;
    i := i + 1
}
```

b) Annotated program $C^{\text{A}}_{\text{Fib}}$

```
x := 1 ;
{x == 1}
y := 0 ;
{x == 1 && y == 0}
i := 1 ;
{x == 1 && y == 0 && i == 1}
while i < n do {i ≤ n && x == Fib(i) && y == Fib(i − 1)}
{
    aux := y ;
{i ≤ n && x == Fib(i) && aux == Fib(i − 1)}
    y := x ;
{i < n && x == Fib(i) && y == Fib(i) && aux == Fib(i − 1)}
    x := x + aux ;
{i ≤ n && x == Fib(i) + Fib(i − 1) && y == Fib(i)}
    i := i + 1 ;
}
```

Fig. 6. Example program: Fibonacci

Example 1 *Consider the program for calculating Fibanocci numbers in Figure 6. The annotated program $C^{\text{A}}_{\text{Fib}}$ shown in b) is the result of annotating C_{Fib} shown in a) in an ad hoc way.*

We extend the assertion language with a vocabulary (Σ, α) of function symbols, where $\alpha : \Sigma \rightarrow \mathbb{N}$ defines the arity of each symbol. From the point of view of verification condition generation, functions are transparent and may occur in the proof obligations. They should be formalized by a set of axioms and rules in the target proof tool. In this particular case, we have $\text{Fib} \in \Sigma$, $\alpha(\text{Fib}) = 1$. *This function* *could be formalized by encoding the following axioms.*

$$\begin{array}{l} \text{Fib}(0) == 0 \\ \text{Fib}(1) == 1 \\ \texttt{Forall}\, x.\, x > 1 \rightarrow \text{Fib}(x) == \text{Fib}(x-1) + \text{Fib}(x-2) \end{array}$$

We also remark the following:

- *The invariant states that variables x and y are used to store respectively the Fibonacci numbers for i and $i-1$, together with an obvious condition regarding a bound for i.*
- *The annotations in the sequence of assignment instructions preceding the loop basically store the current state of the program.*
- *Inside the loop body, the annotated state is reset, since all that is known when execution enters this sequence of instructions in a particular iteration is that the invariant was initially valid. The annotations are then propagated forward from the invariant with each assignment instruction.*
- *The annotations are somewhat optimised. For instance after the first instruction in the loop body, since y is about to lose its present value, there is no need to keep the old value in the annotation. Similarly, aux is dropped from the annotation as soon as it is clear that it will no longer be required.*

Let us consider the following specification for this annotated program $C^{\text{A}}_{\text{Fib}}$

$$\{n > 0\}\, C^{\text{A}}_{\text{Fib}}\, \{x == \text{Fib}(n)\}$$

Suppose we take composition of programs to be right-associative and let C be the body of the loop; then applying the rule for the sequence construct we obtain the following two proof obligations.

$$\begin{array}{l} \{n > 0\} \\ x := 1\, ;\, \{x == 1\}\, y := 0\, ;\, \{x == 1 \;\&\&\; y == 0\}\, i := 1 \\ \{x == 1 \;\&\&\; y == 0 \;\&\&\; i == 1\} \end{array}$$

and

$$\begin{array}{l} \{x == 1 \;\&\&\; y == 0 \;\&\&\; i == 1\} \\ \textbf{while}\ i < n\ \textbf{do}\, \{i \leq n \;\&\&\; x == \text{Fib}(i) \;\&\&\; y == \text{Fib}(i-1)\}\, C \\ \{x == \text{Fib}(n)\} \end{array}$$

The former will generate trivial verification conditions, and for the latter, applying the loop rule yields two verification conditions, namely

$$\models x == 1 \;\&\&\; y == 0 \;\&\&\; i == 1 \rightarrow i \leq n \;\&\&\; x == \text{Fib}(i) \;\&\&\; y == \text{Fib}(i-1)$$

and

$$\models i \leq n \;\&\&\; x == \text{Fib}(i) \;\&\&\; y == \text{Fib}(i-1) \;\&\&\; !\,(i < n) \rightarrow x == \text{Fib}(n)$$

together with the proof obligation corresponding to the loop invariant preservation,

$$\{i \leq n \;\&\&\; x == \text{Fib}(i) \;\&\&\; y == \text{Fib}(i-1) \;\&\&\; i < n\}\, C\, \{i \leq n \;\&\&\; x == \text{Fib}(i) \;\&\&\; y == \text{Fib}(i-1)\}$$

The reader is invited to continue this example through to the end. We will return to it in subsequent sections.

Note that if the annotations introduced in a program are 'wrong' (for instance, the user provides as loop invariant a property that is not preserved by the loop body), there is no risk of corrupting the soundness of the verification process: simply, proof obligations will be created that cannot be proved.

Each choice of annotations leads to the construction of a unique tree in system Hga for a given Hoare triple (possibly a proof tree if the side-conditions are valid). In system Hg for non-annotated programs, many different proof trees are admissible for the same specification. However, if a Hoare triple for an annotated program is derivable in system Hga, then the corresponding triple for the program without annotations is derivable in system Hg. To formally state this property we first define an erasure function $\mathsf{er}_{\mathrm{A}} : \mathbf{AComm} \to \mathbf{Comm}$ inductively as follows

$$\begin{aligned}
\mathsf{er}_{\mathrm{A}}(\mathbf{skip}) &= \mathbf{skip} \\
\mathsf{er}_{\mathrm{A}}(x := e) &= x := e \\
\mathsf{er}_{\mathrm{A}}(C_1; \{R\}\, C_2) &= \mathsf{er}_{\mathrm{A}}(C_1); \mathsf{er}_{\mathrm{A}}(C_2) \\
\mathsf{er}_{\mathrm{A}}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f) &= \mathbf{if}\ b\ \mathbf{then}\ \mathsf{er}_{\mathrm{A}}(C_t)\ \mathbf{else}\ \mathsf{er}_{\mathrm{A}}(C_f) \\
\mathsf{er}_{\mathrm{A}}(\mathbf{while}\ b\ \mathbf{do}\, \{I\}\, C) &= \mathbf{while}\ b\ \mathbf{do}\ \mathsf{er}_{\mathrm{A}}(C)
\end{aligned}$$

Proposition 3 *If* $\vdash_{\mathsf{Hga}} \{P\}\, C\, \{Q\}$, *then* $\vdash_{\mathsf{Hg}} \{P\}\, \mathsf{er}_{\mathrm{A}}(C)\, \{Q\}$.

Proof. By induction on the derivation of $\vdash_{\mathsf{Hga}} \{P\}\, C\, \{Q\}$.

We say that a program C is *correctly annotated with respect to a specification* (P, Q) if $\vdash_{\mathsf{Hga}} \{P\}\, C\, \{Q\}$ whenever $\vdash_{\mathsf{Hg}} \{P\}\, \mathsf{er}_{\mathrm{A}}(C)\, \{Q\}$.

Note that additionally to the sources of errors identified in Section 4, annotations provide a new opportunity for errors to occurs. *Annotation errors* occur when a program has been annotated with assertions that may not hold when the corresponding program point is reached during execution, which may well cause the verification of the program to fail.

It is clear from the above example that annotating programs is a mostly tedious activity. This is true in particular for intermediate assertions in sequences of commands. Fortunately, these assertions can easily be inferred mechanically, if the program is seen in the context of a specification. Consider for instance the Hoare triple

$$\begin{array}{l}
\{x == 5\ \&\&\ y == 10\} \\
\quad aux := y\,; \\
\quad y := x\,; \\
\quad x := x + aux\,; \\
\{x > 10\ \&\&\ y == 5\}
\end{array}$$

One possible correct way to annotate this sequence of commands is to work backwards from the postcondition to obtain a precondition for each atomic command (in this example it suffices to use the assignment rule). The alternative would be to propagate the precondition forward. Figure 7, a) and b), illustrates these two approaches.

a) Backward propagation of the postcondition	b) Forward propagation of the precondition
$\{x == 5\ \&\&\ y == 10\}$	$\{x == 5\ \&\&\ y == 10\}$
$\quad aux := y\,;$	$\quad aux := y\,;$
$\{x + aux > 10\ \&\&\ x == 5\}$	$\{x == 5\ \&\&\ y == 10\ \&\&\ aux == 10\}$
$\quad y := x\,;$	$\quad y := x\,;$
$\{x + aux > 10\ \&\&\ y == 5\}$	$\{x == 5\ \&\&\ y == 5\ \&\&\ aux == 10\}$
$\quad x := x + aux\,;$	$\quad x := x + aux\,;$
$\{x > 10\ \&\&\ y == 5\}$	$\{x > 10\ \&\&\ y == 5\}$

Fig. 7. Annotation propagation

The two annotation strategies exemplified in Figure 7 will in the next two sections be explored and give rise to two different VCGen algorithms, that work on programs annotated with loop invariants only. We shall now state some lemmas concerning such programs.

We let **IComm** denote the class of programs with annotated while commands. Its abstract syntax is defined by

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

$$\frac{}{\{P\}\,\mathbf{skip}\,\{Q\}} \quad \text{if } \models P \to Q \qquad\qquad \frac{}{\{P\}\,x := e\,\{Q\}} \quad \text{if } \models P \to Q[x \mapsto e]$$

$$\frac{\{P\}\,C_1\,\{R\} \qquad \{R\}\,C_2\,\{Q\}}{\{P\}\,C_1\,;\,C_2\,\{Q\}}$$

$$\frac{\{I \mathbin{\&\&} b\}\,C\,\{I\}}{\{P\}\,\mathbf{while}\ b\ \mathbf{do}\,\{I\}\,C\,\{Q\}} \quad \text{if } \models P \to I \ \text{ and } \ \models I \mathbin{\&\&} !\,b \to Q$$

$$\frac{\{P \mathbin{\&\&} b\}\,C_t\,\{Q\} \qquad \{P \mathbin{\&\&} !\,b\}\,C_f\,\{Q\}}{\{P\}\,\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\,\{Q\}}$$

Fig. 8. Goal-directed Hoare logic with annotated while-loops – System Hgi.

An inference system of Hoare logic for **IComm** programs is shown in Figure 8. We call it system Hgi.

Of course, every derivation of a Hoare triple in Hgi has a correspondence in Hg. To state this property formally we define an erasure function $\mathsf{er}_{\mathrm{I}} : \mathbf{IComm} \to \mathbf{Comm}$ similar to er_{A} (with the obvious adaptation).

Proposition 4 *If* $\vdash_{\mathsf{Hgi}} \{P\}\,C\,\{Q\}$*, then* $\vdash_{\mathsf{Hg}} \{P\}\,\mathsf{er}_{\mathrm{I}}(C)\,\{Q\}$*.*

Proof. By induction on the derivation of $\vdash_{\mathsf{Hgi}} \{P\}\,C\,\{Q\}$.

For $C \in \mathbf{IComm}$, $P, Q \in \mathbf{Assert}$, if whenever $\vdash_{\mathsf{Hg}} \{P\}\,\mathsf{er}_{\mathrm{I}}(C)\,\{Q\}$ it holds that $\vdash_{\mathsf{Hgi}} \{P\}\,C\,\{Q\}$, we say that C is *correctly annotated wrt.* (P, Q).

Lemma 3 *If* $\vdash_{\mathsf{Hgi}} \{P\}\,C\,\{Q\}$*, then* C *is correctly annotated wrt.* (P, Q)*.*

Proof. Trivial, regarding the definition of correctly annotated program.

Lemma 4 *If* C *is correctly annotated wrt.* (P, Q) *and both* $\models P' \to P$ *and* $\models Q \to Q'$ *hold, then* C *is correctly annotated wrt.* (P', Q')*.*

Proof. Immediate from Lemma 2.

Lemma 5

1. *If* $\vdash_{\mathsf{Hg}} \{P\}\,\mathsf{er}_{\mathrm{I}}(C_1)\,;\,\mathsf{er}_{\mathrm{I}}(C_2)\,\{Q\}$ *and* $C_1\,;\,C_2$ *is correctly annotated wrt.* (P, Q)*, then, for some assertion* R*,* C_1 *is correctly annotated wrt.* (P, R)*,* C_2 *is correctly annotated wrt.* (R, Q) *and both* $\vdash_{\mathsf{Hg}} \{P\}\,\mathsf{er}_{\mathrm{I}}(C_1)\,\{R\}$ *and* $\vdash_{\mathsf{Hg}} \{R\}\,\mathsf{er}_{\mathrm{I}}(C_2)\,\{Q\}$ *holds.*
2. *If* $\vdash_{\mathsf{Hg}} \{P\}\,\mathbf{if}\ b\ \mathbf{then}\ \mathsf{er}_{\mathrm{I}}(C_t)\ \mathbf{else}\ \mathsf{er}_{\mathrm{I}}(C_f)\,\{Q\}$ *and* $\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f$ *is correctly annotated wrt.* (P, Q)*, then* C_t *is correctly annotated wrt.* $(P \mathbin{\&\&} b, Q)$*,* C_f *is correctly annotated wrt.* $(P \mathbin{\&\&} !\,b, Q)$ *and both* $\vdash_{\mathsf{Hg}} \{P \mathbin{\&\&} b\}\,\mathsf{er}_{\mathrm{I}}(C_t)\,\{Q\}$ *and* $\vdash_{\mathsf{Hg}} \{P \mathbin{\&\&} !\,b\}\,\mathsf{er}_{\mathrm{I}}(C_f)\,\{Q\}$ *holds.*

Proof. Both (1) and (2) are proved by definition of correctly annotated program, case analysis on the rules of system Hgi, Lemma 3 and Prop. 4.

$$
\begin{aligned}
\mathsf{prec}(\mathbf{skip}, Q) &= Q \\
\mathsf{prec}(x := e, Q) &= Q[x \mapsto e] \\
\mathsf{prec}(C_1; C_2, Q) &= \mathsf{prec}(C_1, \mathsf{prec}(C_2, Q)) \\
\mathsf{prec}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, Q) &= (b \to \mathsf{prec}(C_t, Q))\ \&\&\ (\,!\,b \to \mathsf{prec}(C_f, Q)) \\
\mathsf{prec}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ C, Q) &= I
\end{aligned}
$$

Fig. 9. Definition of weakest precondition for **IComm** programs: prec

Gordon [22] has proposed a mechanisation of Hoare logic in the HOL proof assistant, which includes derivations of the inference rules of Hoare logic from a semantic description of the language. A system that is close to system Hga is also proposed.

In Gordon's system these rules are not seen as inference rules; they are instead used to define *tactics* for the prover. A tactic is a function used to advance the proof construction, i.e. it is applied to the current proof state to produce sub-goals of the present goal. In Gordon's system the VCGen is itself implemented as a tactic. The system incorporates notions that will be explained in the next section, so we will return to it.

Gordon uses a so-called *shallow embedding* of the language into the proof system's logic, which precludes proving the assignment axiom as a HOL theorem (it is a meta-level property). Homeier and Martin [27] use a *deep embedding* instead and achieve a proof of correctness for a VCGen; their work very likely reports the first *fully verified VCGen* .

The difference between a shallow and a deep embedding is dictated by the way in which the represented languages (for programs and assertions) are related to the object language of the theorem prover – either as extensions to the latter, or constructed with separate data types. See [3] for a survey on representing Hoare logic in the language of a theorem prover, and the different embedding possibilities.

7 Weakest Preconditions

Backward propagation of assertions can be realized through the use of *weakest preconditions*. Given a program $C \in \mathbf{IComm}$ and a postcondition Q, we calculate an assertion $\mathsf{prec}(C, Q)$ such that $\{\mathsf{prec}(C, Q)\}\, C\, \{Q\}$ is valid and moreover if $\{P\}\, C\, \{Q\}$ is valid for some P then $\models P \to \mathsf{prec}(C, Q)$. Thus $\mathsf{prec}(C, Q)$ is the weakest precondition that grants the truth of postcondition Q after terminating executions of C.

Weakest preconditions are implicit in Hoare logic, in the way that the assignment axiom propagates postconditions backwards: $Q[x \mapsto e]$ is the weakest precondition for Q to hold after execution of $x := e$. The explicit definition of prec given in Figure 9 propagates postconditions backwards through the remaining program constructs.

In the sequence command, the weakest precondition of the second command is fed as postcondition to the first command, to obtain the precondition of the combined command. The clause for conditional is also straightforward to understand: the weakest precondition of such a command is the weakest precondition of the appropriate branch (calculated considering the same postcondition), depending on the value of the boolean condition.

In general, since the number of iterations may not be known at compile time, the weakest precondition of a loop cannot be calculated statically – performing a one-step expansion of a *while* command (using conditional) and trying to derive the

$$
\begin{aligned}
\mathsf{VC}(\{P\}\,\mathbf{skip}\,\{Q\}) &= \{[\; P \to Q \;]\} \\
\mathsf{VC}(\{P\}\,x := e\,\{Q\}) &= \{[\; P \to Q[x \mapsto e] \;]\} \\
\mathsf{VC}(\{P\}\,C_1\,;\,C_2\,\{Q\}) &= \mathsf{VC}(\{P\}\,C_1\,\{\mathsf{prec}(C_2,Q)\}) \cup \mathsf{VC}(\{\mathsf{prec}(C_2,Q)\}\,C_2\,\{Q\}) \\
\mathsf{VC}(\{P\}\,\mathbf{while}\;b\;\mathbf{do}\,\{I\}\,C\,\{Q\}) &= \{[\; P \to I \;],[\; I \;\&\&\; !\,b \to Q \;]\} \cup\; \mathsf{VC}(\{I \;\&\&\; b\}\,C\,\{I\}) \\
\mathsf{VC}(\{P\}\,\mathbf{if}\;b\;\mathbf{then}\;C_t\;\mathbf{else}\;C_f\,\{Q\}) &= \mathsf{VC}(\{P \;\&\&\; b\}\,C_t\,\{Q\}) \cup \mathsf{VC}(\{P \;\&\&\; !\,b\}\,C_f\,\{Q\})
\end{aligned}
$$

Fig. 10. A VCGen for **IComm** based on weakest preconditions: VC

$$
\begin{array}{ll}
C' \left\{ \begin{array}{l}
\qquad A_0 = \{1 \le n \;\&\&\; 1 == \mathrm{Fib}(1) \;\&\&\; 0 == \mathrm{Fib}(1-1)\} \\
x := 1\,; \\
\qquad A_1 = \{1 \le n \;\&\&\; x == \mathrm{Fib}(1) \;\&\&\; 0 == \mathrm{Fib}(1-1)\} \\
y := 0\,; \\
\qquad A_2 = \{1 \le n \;\&\&\; x == \mathrm{Fib}(1) \;\&\&\; y == \mathrm{Fib}(1-1)\} \\
i := 1\,; \\
\qquad I = \{i \le n \;\&\&\; x == \mathrm{Fib}(i) \;\&\&\; y == \mathrm{Fib}(i-1)\}
\end{array} \right. \\
\quad \mathbf{while}\; i < n \;\mathbf{do}\,\{\; i \le n \;\&\&\; x == \mathrm{Fib}(i) \;\&\&\; y == \mathrm{Fib}(i-1)\} \\
C'' \left\{ \begin{array}{l}
\{ \\
\quad aux := y\,; \\
\qquad\qquad A_3 = \{i+1 \le n \;\&\&\; x + aux == \mathrm{Fib}(i+1) \;\&\&\; x == \mathrm{Fib}((i+1)-1)\} \\
\quad y := x\,; \\
\qquad\qquad A_4 = \{i+1 \le n \;\&\&\; x + aux == \mathrm{Fib}(i+1) \;\&\&\; y == \mathrm{Fib}((i+1)-1)\} \\
\quad x := x + aux\,; \\
\qquad\qquad A_5 = \{i+1 \le n \;\&\&\; x == \mathrm{Fib}(i+1) \;\&\&\; y == \mathrm{Fib}((i+1)-1)\} \\
\quad i := i+1\,; \\
\}
\end{array} \right.
\end{array}
$$

Fig. 11. Fibonacci with annotated loop: $C^{\mathrm{I}}_{\mathrm{Fib}}$

weakest precondition from that expansion leads to a recursive equation. However, for the case of loops annotated with invariants, the weakest precondition can be defined to be precisely the invariant assertion. The reason for this is that all the reasoning about the behaviour of loops depends on the invariants being granted as preconditions. In an annotated program, the invariant of a loop is thus the weakest precondition required for any postcondition to hold.

Proposition 5 *If* $C \in$ **IComm** *is correctly annotated wrt.* (P, Q) *and* $\vdash_{\mathsf{Hg}} \{P\}\,\mathsf{er}_{\mathrm{I}}(C)\,\{Q\}$*, then*

1. $\vdash_{\mathsf{Hgi}} \{\mathsf{prec}(C,Q)\}\,C\,\{Q\}$
2. $\models P \to \mathsf{prec}(C,Q)$

Proof. By induction on the structure of C, and using lemmas 2, 3 and 5 for (1) and 2, 4 and 5 for (2).

Figure 10 contains the straightforward definition of a VCGen obtained from system Hgi by using prec as an auxiliary function. The following lemma states that the verification conditions produced by this algorithm, for a given Hoare triple, are exactly the side conditions of a derivation tree of that triple in system Hgi.

Lemma 6 *For any* $C \in$ **IComm**, $\mathsf{VC}(\{P\}\,C\,\{Q\}) \Vdash_{\mathsf{Hgi}} \{P\}\,C\,\{Q\}$

Proof. By induction on the structure of C.

Before looking at an example, we remark the following. The VCGen is implicitly constructing a derivation of Hg following a fixed strategy. This consists in always constructing the sub-derivation corresponding to the second sub-command in any sequence command, until the intermediate condition is attained, at which point

$$
\begin{array}{l}
\mathsf{VC}(\{n > 0\}\, C^{\mathrm{I}}_{\mathrm{Fib}}\, \{x == \mathrm{Fib}(n)\}) \\
= \mathsf{VC}(\{n > 0\}\, x := 1\, \{A_1\}) \;\cup\; \mathsf{VC}(\{A_1\}\, y := 0\, \{A_2\}) \;\cup\; \mathsf{VC}(\{A_2\}\, i := 1\, \{I\}) \;\cup \\
\quad \mathsf{VC}(\{I\}\, \textbf{while}\ i < n\ \textbf{do}\, \{I\}\, C''\, \{x == \mathrm{Fib}(n)\}) \\
= \{[\, n > 0 \rightarrow 1 \leq n \,\&\&\, 1 == \mathrm{Fib}(1) \,\&\&\, 0 == \mathrm{Fib}(1-1)\,]\} \;\cup \\
\quad \{[\, 1 \leq n \,\&\&\, x == \mathrm{Fib}(1) \,\&\&\, 0 == \mathrm{Fib}(1-1) \rightarrow 1 \leq n \,\&\&\, x == \mathrm{Fib}(1) \,\&\&\, 0 == \mathrm{Fib}(1-1)\,]\} \;\cup \\
\quad \{[\, 1 \leq n \,\&\&\, x == \mathrm{Fib}(1) \,\&\&\, y == \mathrm{Fib}(1-1) \rightarrow 1 \leq n \,\&\&\, x == \mathrm{Fib}(1) \,\&\&\, y == \mathrm{Fib}(1-1)\,]\} \;\cup \\
\quad \{[\, I \rightarrow I\,], [\, I \,\&\&\, !\,(i < n) \rightarrow x == \mathrm{Fib}(n)\,]\} \;\cup\; \mathsf{VC}(\{I \,\&\&\, i < n\}\, C''\, \{I\}) \\
= \{[\, n > 0 \rightarrow 1 \leq n \,\&\&\, 1 == \mathrm{Fib}(1) \,\&\&\, 0 == \mathrm{Fib}(1-1)\,], \\
\quad [\, 1 \leq n \,\&\&\, x == \mathrm{Fib}(1) \,\&\&\, 0 == \mathrm{Fib}(1-1) \rightarrow 1 \leq n \,\&\&\, x == \mathrm{Fib}(1) \,\&\&\, 0 == \mathrm{Fib}(1-1)\,], \\
\quad [\, 1 \leq n \,\&\&\, x == \mathrm{Fib}(1) \,\&\&\, y == \mathrm{Fib}(1-1) \rightarrow 1 \leq n \,\&\&\, x == \mathrm{Fib}(1) \,\&\&\, y == \mathrm{Fib}(1-1)\,], \\
\quad [\, I \rightarrow I\,], [\, I \,\&\&\, !\,(i < n) \rightarrow x == \mathrm{Fib}(n)\,], \\
\quad [\, I \,\&\&\, i < n \rightarrow i + 1 \leq n \,\&\&\, x + y == \mathrm{Fib}(i+1) \,\&\&\, x == \mathrm{Fib}((i+1)-1)\,], \\
\quad [\, A_3 \rightarrow A_3\,], [\, A_4 \rightarrow A_4\,], [\, A_5 \rightarrow A_5\,]\}
\end{array}
$$

Fig. 12. Fibonacci example: verification conditions obtained with VC.

the first sub-derivation can then be constructed. This results in the derivation that would be constructed in Hga if the program was previously annotated with intermediate conditions calculated by prec.

Example 2 *Figure 12 shows the application of the VCGen algorithm to our running example program. The intermediate conditions (generated as weakest preconditions by* prec*) are shown in Figure 11 as annotations in the program, for the sake of readability.*

Observe that while all the conditions generated are straightforward to prove, most are tautological, of the form $[\, Q \rightarrow Q\,]$ *for some* Q*. The ones that are not are*

1. $[\, n > 0 \rightarrow 1 \leq n \,\&\&\, 1 == \mathrm{Fib}(1) \,\&\&\, 0 == \mathrm{Fib}(1-1)\,]$*, which states that the precondition in the specification is stronger than the weakest precondition calculated for the program;*
2. $[\, I \,\&\&\, i < n \rightarrow i + 1 \leq n \,\&\&\, x + y == \mathrm{Fib}(i+1) \,\&\&\, x == \mathrm{Fib}((i+1)-1)\,]$*, which corresponds to the preservation of the loop invariant; and*
3. $[\, I \,\&\&\, !\,(i < n) \rightarrow x == \mathrm{Fib}(n)\,]$*, corresponding to the "use case" upon termination of the loop, which must imply the postcondition in the Hoare triple.*

The reason why this VCGen generates many tautological conditions is that for any command of the form $C_1\,;\, x := e$ with postcondition Q, a precondition $\mathsf{prec}(x := e, Q)$ will be generated for the assignment command. Conditions like the following will proliferate

$$\mathsf{VC}(\{\mathsf{prec}(x := e, Q)\}\, x := e\, \{Q\}) = [\, Q[x \mapsto e] \rightarrow Q[x \mapsto e]\,]$$

It is however possible to define a VCGen that eliminates these unnecessary conditions. The trick is to calculate verification conditions independently of preconditions. The VCGen in Figure 13, that we call VCG, uses an auxiliary recursive function $\mathsf{VC}_{\mathsf{aux}}$ that does exactly this.

Note that for any sequence of assignments VCG generates a sole verification condition. This is a consequence of the following lemma.

Lemma 7 $\mathsf{VC}_{\mathsf{aux}}(C_1\,;\, C_2\,;\, \ldots\,;\, C_n, Q) = \emptyset$*, if each* C_i *is either a skip or an assignment.*

Proof. By induction on the length of the sequence $C_1\,;\, C_2\,;\, \ldots\,;\, C_n$.

The verification conditions for a Hoare triple $\{P\}\, C\, \{Q\}$ are then calculated by adding to the set $\mathsf{VC}_{\mathsf{aux}}(C, Q)$ a condition explicitly relating P and the weakest precondition $\mathsf{prec}(C, Q)$. This may be seen as the principal verification condition,

$$
\begin{aligned}
\mathsf{VC_{aux}}(\mathbf{skip}, Q) &= \emptyset \\
\mathsf{VC_{aux}}(x := e, Q) &= \emptyset \\
\mathsf{VC_{aux}}(C_1 ; C_2, Q) &= \mathsf{VC_{aux}}(C_1, \mathsf{prec}(C_2, Q)) \ \cup\ \mathsf{VC_{aux}}(C_2, Q) \\
\mathsf{VC_{aux}}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, Q) &= \mathsf{VC_{aux}}(C_t, Q) \ \cup\ \mathsf{VC_{aux}}(C_f, Q) \\
\mathsf{VC_{aux}}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ C, Q) &= \{[\ (I \ \&\&\ b) \to \mathsf{prec}(C, I)\ ]\} \ \cup\ \mathsf{VC_{aux}}(C, I) \ \cup\ \{[\ (I \ \&\&\ !\, b) \to Q\ ]\} \\
\\
\mathsf{VCG}(\{P\}\, C\, \{Q\}) &= \{[\ P \to \mathsf{prec}(C, Q)\ ]\} \ \cup\ \mathsf{VC_{aux}}(C, Q)
\end{aligned}
$$

Fig. 13. An improved VCGen: VCG

$$
\begin{aligned}
&\mathsf{VCG}(\{n > 0\}\, C^{\mathrm{I}}_{\mathrm{Fib}}\, \{x == \mathrm{Fib}(n)\}) \\
&= \{[\ n > 0 \to \mathsf{prec}(C^{\mathrm{I}}_{\mathrm{Fib}}, x == \mathrm{Fib}(n))\ ]\} \ \cup\ \mathsf{VC_{aux}}(C^{\mathrm{I}}_{\mathrm{Fib}}, x == \mathrm{Fib}(n)) \\
&= \{[\ n > 0 \to 1 \leq n \ \&\&\ 1 == \mathrm{Fib}(1) \ \&\&\ 0 == \mathrm{Fib}(1-1)\ ]\} \ \cup\ \mathsf{VC_{aux}}(C', I) \ \cup \\
&\quad \mathsf{VC_{aux}}(\mathbf{while}\ i < n\ \mathbf{do}\ \{I\}\ C'', x == \mathrm{Fib}(n)) \\
&= \{[\ n > 0 \to 1 \leq n \ \&\&\ 1 == \mathrm{Fib}(1) \ \&\&\ 0 == \mathrm{Fib}(1-1)\ ]\} \ \cup\ \emptyset \ \cup \\
&\quad \{[\ I \ \&\&\ i < n \to \mathsf{prec}(C'', I)\ ]\} \ \cup\ \mathsf{VC_{aux}}(C'', I) \ \cup\ \{[\ I \ \&\&\ !\,(i < n) \to x == \mathrm{Fib}(n)\ ]\} \\
&= \{[\ n > 0 \to 1 \leq n \ \&\&\ 1 == \mathrm{Fib}(1) \ \&\&\ 0 == \mathrm{Fib}(1-1)\ ], \\
&\quad [\ I \ \&\&\ i < n \to i + 1 \leq n \ \&\&\ x + y == \mathrm{Fib}(i+1) \ \&\&\ x == \mathrm{Fib}((i+1)-1)\ ], \\
&\quad [\ I \ \&\&\ !\,(i < n) \to x == \mathrm{Fib}(n)\ ]\}
\end{aligned}
$$

Fig. 14. Fibonacci example, using VCG.

obtained from the backward propagation of the postcondition, whereas $\mathsf{VC_{aux}}(C, Q)$ calculates secondary verification conditions resulting from loops.

It is easy to see that, for a given Hoare triple, the verification conditions generated by VCG entail the ones generated by VC.

Lemma 8 *If* $\models \mathsf{VCG}(\{P\}\, C\, \{Q\})$*, then* $\models \mathsf{VC}(\{P\}\, C\, \{Q\})$*.*

Proof. By induction on the structure of C.

Proposition 6 (Correctness of VCG) *Let* $C \in \mathbf{IComm}$ *and* $P, Q \in \mathbf{Assert}$ *such that* $\models \mathsf{VCG}(\{P\}\, C\, \{Q\})$*. Then* $\vdash_{\mathsf{Hg}} \{P\}\, \mathsf{er_I}(C)\, \{Q\}$

Proof. Immediate by lemmas 6 and 8, and proposition 4.

Figure 14 illustrates the use of this VCGen. We remark that, since prec and $\mathsf{VC_{aux}}$ perform similar traversals of the program structure, they could be fused into a single function using the well-known *tupling* technique of functional programming.

Bertot's Coq development on program semantics [10] (see Section 3) includes both the definition of the VC as a recursive function, and its proof of correctness.

Gordon's VCGen [22] (see Section 6) is very similar to the VCG presented in this section; the difference is that his algorithm does not require invoking an external function for calculating weakest preconditions; instead sequence commands are required to be partially annotated. A sequence command of the form $C_1 ; x := e$ does not need an annotation, but all other sequences have to be annotated with an intermediate assertion, as in $C_1 ;\ \{R\}\ C_2$. This system thus stands between Hga and Hgi.

Weakest preconditions were introduced by Dijkstra [17] from a semantic perspective. The idea was to interpret programs as *predicate transformers* – mappings of postconditions into preconditions. Dijkstra used a more abstract programming language based on *guarded commands*, and in fact some modern and very advanced tools for program verification are based on such a language. This will be the topic of Section 8.

A Note on Auxiliary Variables. There is a problem with the specification of Fibonacci that we have been using as a running example. The specification $(n > 0, x == \mathrm{Fib}(n))$ has trivial solutions, since nothing prevents the program from modifying the value of the input variable n. Thus the triple $\{n > 0\}\, n := 0 \,;\, x := 0\, \{x == \mathrm{Fib}(n)\}$ is a valid Hoare triple.

The problem can be avoided with the use of an *auxiliary variable* to record the initial value of n. The triple

$$\{n > 0 \;\&\&\; n == n_0\}\, F\, \{x == \mathrm{Fib}(n) \;\&\&\; n == n_0\}$$

solves the problem as long as n_0 is indeed auxiliary, i.e. it is not used as a program variable in F. Hoare logic has no explicit support for auxiliary variables: it is not possible to write the specification above in a way that formally prevents n_0 from being modified by F. This is particularly relevant if one is interested in doing modular verification, since one may have to reason about a program that calls an external procedure for which only the specification, and not the code, is available (see Section 10.3 below).

This limitation has since been addressed at the theoretical level [35, 49, 51]. In the context of program verification tools a practical solution has been found, which consists in enriching the assertion language with the means to refer to the value of an expression at a given program point, and introducing fresh variables accordingly.

For instance, our example specification could be written

$$\{n > 0\}\, F\, \{x == \mathrm{Fib}(n) \;\&\&\; n == \mathsf{old}(n)\}$$

or simply

$$\{n > 0\}\, F\, \{x == \mathrm{Fib}(\mathsf{old}(n))\}$$

where $\mathsf{old}(n)$ refers to the value of n in the initial state, in which the precondition is *true*. The generation of verification conditions can be altered to cope with this operator in two ways, that we illustrate by means of an example.

The first method would be to interpret the $\mathsf{old}(\cdot)$ operator as simply acting as a protection against variable substitution. The definition of substitution could be extended as follows:

$$\mathsf{old}(e)\,[x \mapsto e'] = \mathsf{old}(e)$$

Thus $\mathsf{old}(e)$ would be treated as a constant. Occurrences of the operator could then be removed from assertions when it is no longer necessary (a task for the $[\cdot]$ operator when exporting verification conditions). Used with the Hoare triple $\{n > 0\}\, C^{\mathrm{I}}_{\mathrm{Fib}}\, \{x == \mathrm{Fib}(\mathsf{old}(n))\}$ this would yield the same verification conditions as before, but it would still work correctly if $C^{\mathrm{I}}_{\mathrm{Fib}}$ modified the value of n.

The second approach would be to introduce in the precondition an equation $\mathsf{old}(e) = x'$ where x' is a fresh auxiliary variable (with respect to the current Hoare triple). This would require modifying the VCGen as follows[3]

$$\begin{aligned}\mathsf{VCG}(\{P\}\, C\, \{Q\}) = \{[\,(P \;\&\&\; e == x') \to \mathsf{prec}(C, Q[\mathsf{old}(e) \mapsto x'])\,]\} \cup\\ \mathsf{VC_{aux}}(C, Q[\mathsf{old}(e) \mapsto x'])\end{aligned}$$

Thus for the above example one would have

$$\begin{aligned}&\mathsf{VCG}(\{n > 0\}\, C^{\mathrm{I}}_{\mathrm{Fib}}\, \{x == \mathrm{Fib}(\mathsf{old}(n))\}) =\\ &\qquad [\, n > 0 \;\&\&\; n == n_0 \to \mathsf{prec}(C^{\mathrm{I}}_{\mathrm{Fib}}, x == \mathrm{Fib}(n_0))\,] \cup\\ &\qquad \mathsf{VC_{aux}}(C^{\mathrm{I}}_{\mathrm{Fib}}, x == \mathrm{Fib}(n_0))\end{aligned}$$

The treatment above (where we admit a single use of old) can easily be adapted to this general case. In general each application of the operator would introduce a new fresh variable.

[3] Here $Q[\mathsf{old}(e) \mapsto x']$ stands for the result of replacing x' for the occurrences of $\mathsf{old}(e)$ in Q.

8 Guarded Commands

We take a detour here to briefly consider a variant of Dijkstra's guarded commands language and Weakest Precondition Calculus (WPC). The calculus provides a verification conditions generator, and guarded commands, although apparently very abstract, are used as intermediate language by at least two standard program verification tools.

The language is different from the programming language we have been considering in a number of ways. First, there is no distinction between boolean expressions and assertions. The language contains two primitives that test the value of assertions: the commands **assert** b and **assume** b both behave like **skip** if b evaluates to *true*. The difference is that the former *terminates abruptly* if b evaluates to *false*, whereas the latter *cannot be executed*. It is thus a partial command that can be used as a *guard* for the execution of a subsequent command.

Another aspect is the presence of a *non-deterministic choice* operator. The command $C_1 \mathbin{[\!]} C_2$ will arbitrarily execute either C_1 or C_2.

The expression $wp.C.Q$ denotes the weakest precondition such that Q holds as a postcondition if the command C terminates. The following calculus defines the weakest precondition semantics of the language.

$$\begin{aligned}
wp.(\textbf{assert}\, b).Q &= b \,\&\&\, Q \\
wp.(\textbf{assume}\, b).Q &= b \to Q \\
wp.(x := e).Q &= Q[x \mapsto e] \\
wp.(C_1; C_2).Q &= wp.C_1.(wp.C_2.Q) \\
wp.(C_1 \mathbin{[\!]} C_2).Q &= wp.C_1.Q \,\&\&\, wp.C_2.Q
\end{aligned}$$

Given a program C, the verification condition generated for the partial correctness Hoare triple $\{P\}\, C\, \{Q\}$ is given by

$$\mathsf{VC}(\{P\}\, C\, \{Q\}) = [\ P \to wp.C.Q\ ]$$

In fact, since it corresponds to partial correctness, this notion is usually known as the weakest *liberal* precondition.

Two crucial constructs of an imperative language – conditional and loops – are missing from the guarded commands language, but they can be encoded, i.e. translated into guarded commands that generate the appropriate weakest preconditions. The command **if** b **then** C_t **else** C_f can be encoded as

$$(\textbf{assume}\, b\,;\, C_t) \mathbin{[\!]} (\textbf{assume}\ !\, b\,;\, C_f)$$

where b and $!\, b$ are used as guards in a choice command, which has the effect of removing non-determinism since it is certain that one of the commands cannot be executed.

Loops can be encoded in a number of ways. If one is willing to give up on soundness and check that the invariant holds for only a limited number of iterations, the loop **while** b **do** $\{I\}\, C$ can simply be encoded as follows

$$\begin{array}{l}
\textbf{assert}\, I\,; \\
(\textbf{assume}\, b\,;\, C\,;\, \textbf{assert}\, I\,;\, \textbf{assume}\, \texttt{false}) \mathbin{[\!]} (\textbf{assume}\ !\, b)
\end{array}$$

Here the loop is taken to execute at most once but this can easily be expanded to an arbitrary fixed number of iterations. The effect of the first **assert** command is to test the invariant in the initial conditions; the first branch of the choice command tests its preservation by an iteration of the loop body; the second branch establishes the falsity of the loop condition on exit. Note that if the choice selects the first branch, the second branch will inevitably be selected when **assume** `false` is reached (the command in the first branch cannot be executed in its entirety), thus this encodes a loop that iterates at most once.

To test the preservation of the invariant in an *arbitrary* iteration of the loop requires to identify all the variables (say $x_1, \ldots, x_n$) assigned in the loop body, and to assign them arbitrary values. Fresh variables $(y_1, \ldots, y_n)$ can be used to this effect. The following translation of the loop first tests that the invariant is initially true, then resets the values of variables, and assumes the truth of the invariant for the current arbitrary state. The choice command that follows is the same as before, but the first branch is now testing the preservation of the invariant in an arbitrary iteration.

$$\begin{array}{l} \textbf{assert}\, I\,; \\ x_1 := y_1\,;\, \ldots\,;\, x_n := y_n\,; \\ \textbf{assume}\, I\,; \\ (\textbf{assume}\, b\,;\, C\,;\, \textbf{assert}\, I\,;\, \textbf{assume}\, \texttt{false}) \,[]\, (\textbf{assume}\, !\, b) \end{array}$$

The specific guarded command language presented in this section has been widely used as a core language in the development of at least two major tools for checking the behaviour of programs: the ESC [41, 39, 14] family of tools (of which ESC/Modula3, ESC/Java and ESC/Java2 are instances) and more recently Boogie [5], a generic VCGen that is being used notably with the Spec# language. Both tools are capable of generating verification conditions as proof obligations for the Simplify [16] prover, but Boogie supports more recent and advanced proof tools.

ESC stands for "extended static checking"; its emphasis was more on providing programmers with tools that could find common errors (such as null dereferencing) rather than on program verification. Boogie is a very sophisticated tool that integrates advanced features such as automatic inference of loop invariants. It is a generic VCGen in the sense that different programming languages can be translated into the BoogiePL language (in a sound way with respect to the generated verification conditions). VC generation based on weakest preconditions has been advanced with the development of Boogie to cover, for instance, programs containing both loops and **goto** statements [6].

Both tools can be used with complex annotation languages for real-world programs. For instance ESC/Java uses JML (a standard annotation language for Java programs, see Section 11), and Boogie has been used as a VCGen for Spec# (similar to JML but for C# programs). In both tools the guarded command language is used as an intermediate language into which source code is translated; verification conditions are generated by applying the weakest-precondition calculus.

An important result was proposed as part of the development of ESC. The simple definition of weakest precondition given above generates VCs whose size is potentially exponential in the size of the source code. Two clauses in the definition of *wp* are responsible for this: the case of the assignment command $x := e$, which may have to create as many copies of the expression e as there are occurrences of x in the postcondition Q; and the choice command, which duplicates Q.

This problem can be fixed [20, 40] by using a two-stage algorithm that produces VCs that are worst-case quadratic in size (and usually close to linear). The idea is that guarded commands are first translated into a *passive form*, where assignments are eliminated, replaced by *assume* commands and fresh variables for each assigned variable. For instance, $x := x + 1; \textbf{assert}\, x > 0$ becomes $\textbf{assume}\, x_1 = x+1\,;\ \textbf{assert}\, x_1 > 0$. This translation preserves the weakest precondition semantics, and while it may increase the size of the code, this growth is worst-case quadratic and near linear in practice.

Passive commands may not affect the state of programs since they do not contain assignment statements. The semantics of two programs may differ only with respect to termination: programs may terminate normally or erroneously. The weakest preconditions $wp.C.true$ and $wp.C.false$ characterize respectively the states from which C may not terminate erroneously and the states from which C may not terminate normally. The weakest precondition for a given postcondition Q can then be written as

$$wp.C.Q = wp.C.true \;\&\&\; (\,!\, wp.C.false \rightarrow Q)$$

The resulting condition is worst-case quadratic in the size of the code.

The weakest precondition technique is quite flexible. For instance it is easy to treat erroneous termination explicitly by calculating weakest preconditions with respect to two postconditions. The following extended definition ensures a given postcondition Q if the program terminates normally and a possibly distinct postcondition R if it terminates abruptly with a failed **assert** command.

$$\begin{aligned}
wp.(\mathbf{assert}\, b).(Q,R) &= (b \;\&\&\; Q) \,\|\, (\,!\, b \;\&\&\; R)\\
wp.(\mathbf{assume}\, b).(Q,R) &= b \rightarrow Q\\
wp.(x := e).(Q,R) &= Q[x \mapsto e]\\
wp.(C_1; C_2).(Q,R) &= wp.C_1.(wp.C_2.(Q,R), R)\\
wp.(C_1 \,[\!]\, C_2).(Q,R) &= wp.C_1.(Q,R) \;\&\&\; wp.C_2.(Q,R)
\end{aligned}$$

Exceptional termination can also be considered in this framework, which inspires our treatment of exceptions for the While language in Section 10.1.

The weakest preconditions calculus has also been studied extensively and applied in many textbooks [17, 34, 23] as a tool for the development of *correct-by-construction* software – quite a different perspective from program verification.

9 Hoare Logic with Updates

In Section 6 it was shown that the intermediate assertions required by the sequencing rule of Hoare logic could be calculated either backwards from postconditions or in a forward fashion starting from the preconditions. In order to define an inference system that propagates assertions in a forward manner, it is convenient to modify the abstract syntax of programs to the following linear version, in which programs are defined as (possibly empty) sequences of commands. We let **PROG** denote the class of programs.

$$\begin{aligned}
\mathbf{PROG} \ni W &::= C \,;\, W \mid \varepsilon\\
\mathbf{ICOMM} \ni C &::= \mathbf{skip} \mid x := e \mid \mathbf{if}\; b\; \mathbf{then}\; W\; \mathbf{else}\; W \mid \mathbf{while}\; b\; \mathbf{do}\, \{A\}\, W
\end{aligned}$$

This corresponds simply to a left-associative view of the sequence operator. We define a translation function $\mathcal{T} : \mathbf{IComm} \rightarrow \mathbf{PROG}$ as follows

$$\begin{aligned}
\mathcal{T}(\mathbf{skip}) &= \mathbf{skip}\,;\, \varepsilon\\
\mathcal{T}(x := e) &= x := e\,;\, \varepsilon\\
\mathcal{T}(C_1\,;\, C_2) &= \mathcal{T}(C_1) @ \mathcal{T}(C_2)\\
\mathcal{T}(\mathbf{if}\; b\; \mathbf{then}\; C_t\; \mathbf{else}\; C_f) &= \mathbf{if}\; b\; \mathbf{then}\; \mathcal{T}(C_t)\; \mathbf{else}\; \mathcal{T}(C_f)\,;\, \varepsilon\\
\mathcal{T}(\mathbf{while}\; b\; \mathbf{do}\, \{I\}\, C) &= \mathbf{while}\; b\; \mathbf{do}\, \{I\}\, \mathcal{T}(C)\,;\, \varepsilon
\end{aligned}$$

where $\cdot @ \cdot$ denotes sequence concatenation.

A natural semantics for programs and commands can be given via a mutually dependent definition of the evaluation relation, similar to what is done in Figure 2 (with the obvious adaptations).

The formulation of specifications that will be defined for these programs uses a notion of *update*. An update is simply a partial mapping from variables to expressions of the language. We write $\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ for the update mapping x_i to e_i, for $i \in \{1, \ldots, n\}$. Let $\mathcal{U} = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ be an update, and t an expression (resp. assertion). We write $\mathcal{U}(t)$ to denote the expression (resp. assertion) obtained by the simultaneous substitution of terms e_i for the free occurrences of variables x_i in t.

Updates are modified by an operation that sets the value of a variable, defined as

$$(\mathcal{U}\,;x := e) = \mathcal{U} \oplus \{x \mapsto \mathcal{U}(e)\}$$

where $\oplus$ denotes function overriding. For two partial functions $f, g : X \hookrightarrow Y$ we define $f \oplus g : X \hookrightarrow Y$ as follows

$$(f \oplus g)\,(x) = \begin{cases} g(x) & \text{if } \ x \in \mathsf{dom}(g) \\ f(x) & \text{if } \ x \notin \mathsf{dom}(g) \wedge x \in \mathsf{dom}(f) \end{cases}$$

The notion of specification will now be slightly modified to correspond to a Hoare triple extended with an update, written $\{P\}[\mathcal{U}]\,W\,\{Q\}$. The idea is that the update is applied to the initial state, in which the precondition holds, before the program is executed. The semantic interpretation of such a specification is

$$[\![\{P\}[\mathcal{U}]\,W\,\{Q\}]\!] = \forall s, s' \in \Sigma.\ [\![P]\!](s) \wedge (W, s_{\mathcal{U}}) \Downarrow s' \Rightarrow [\![Q]\!](s')$$

where the updated state $s_{\mathcal{U}}$ is defined as follows:

$$s_{\mathcal{U}}\,(x) = \begin{cases} s(x) & \text{if } \ x \notin \mathsf{dom}(\mathcal{U}) \\ [\![\mathcal{U}(x)]\!](s) & \text{if } \ x \in \mathsf{dom}(\mathcal{U}) \end{cases}$$

Before introducing the inference system we proceed with some lemmas involving the concepts described.

Lemma 9 *For all $C \in$ **IComm**, $(\mathcal{T}(C), s) \Downarrow s'$ iff $(C, s) \Downarrow s'$*

Proof. By induction on the structure of C.

Proposition 7 *For $C \in$ **IComm**, $[\![\{P\}[\emptyset]\,\mathcal{T}(C)\,\{Q\}]\!] = [\![\{P\}\,C\,\{Q\}]\!]$*

Proof. Immediate by Lemma 9.

Lemma 10 *$(W_1 @ W_2, s) \Downarrow s'$ iff $(W_1, s) \Downarrow s''$ and $(W_2, s'') \Downarrow s'$, for some $s'' \in \Sigma$*

Proof. By induction on the length of W_1.

Lemma 11 $[\![\mathcal{U}(Q)]\!](s) = [\![Q]\!](s_{\mathcal{U}})$

Proof. By induction on the structure of Q.

Lemma 12 *1.* $s_{(\mathcal{U}\,;\,x:=e)} = (s_{\mathcal{U}})_{x:=e}$
2. $(W, s_{(\mathcal{U}\,;\,x:=e)}) \Downarrow s'$ *iff* $(x := e\,;\,W, s_{\mathcal{U}}) \Downarrow s'$

Proof. Directly by unfolding the definitions.

The inference system of Hoare logic with updates is given in Figure 15. We name it system Hu. There is one rule for each program construct, except for sequencing, since all programs are now seen as sequences. The rule for the empty program (called the *exit* rule) introduces a verification condition; the remaining rules are selected by pattern-matching on the first command of the program. So, the construction of derivation trees is syntax-directed.

Proposition 8 (Soundness) *Every specification derivable in system* Hu *is semantically valid: If $\vdash_{\mathsf{Hu}} \{P\}[\mathcal{U}]\,W\,\{Q\}$, then $[\![\{P\}[\mathcal{U}]\,W\,\{Q\}]\!] = true$.*

Proof. By induction on the derivation of $\vdash_{\mathsf{Hu}} \{P\}[\mathcal{U}]\,W\,\{Q\}$.

$$\frac{}{\{P\}[\mathcal{U}]\,\varepsilon\,\{Q\}} \quad \text{if } \models P \to \mathcal{U}(Q)$$

$$\frac{\{P\}[\mathcal{U}]\,W\,\{Q\}}{\{P\}[\mathcal{U}]\,\textbf{skip}\,;\,W\,\{Q\}} \qquad \frac{\{P\}[\mathcal{U}\,;x:=e]\,W\,\{Q\}}{\{P\}[\mathcal{U}]\,x:=e\,;\,W\,\{Q\}}$$

$$\frac{\{I\ \&\&\ b\}[\emptyset]\,W_t\,\{I\} \qquad \{I\ \&\&\ !\,b\}[\emptyset]\,W\,\{Q\}}{\{P\}[\mathcal{U}]\,\textbf{while}\ b\ \textbf{do}\,\{I\}\,W_t\,;\,W\,\{Q\}} \quad \text{if } \models P \to \mathcal{U}(I)$$

$$\frac{\{P\ \&\&\ \mathcal{U}(b)\}[\mathcal{U}]\,W_t@W\,\{Q\} \qquad \{P\ \&\&\ !\,\mathcal{U}(b)\}[\mathcal{U}]\,W_f@W\,\{Q\}}{\{P\}[\mathcal{U}]\,\textbf{if}\ b\ \textbf{then}\ W_t\ \textbf{else}\ W_f\,;\,W\,\{Q\}}$$

Fig. 15. Rules of Hoare logic with updates – System Hu

$$\begin{aligned}
\mathsf{VCGu}(\{P\}[\mathcal{U}]\,\textbf{skip}\,;\,W\,\{Q\}) &= \mathsf{VCGu}(\{P\}[\mathcal{U}]\,W\,\{Q\})\\
\mathsf{VCGu}(\{P\}[\mathcal{U}]\,x:=e\,;\,W\,\{Q\}) &= \mathsf{VCGu}(\{P\}[\mathcal{U}\,;\,x:=e]\,W\,\{Q\})\\
\mathsf{VCGu}(\{P\}[\mathcal{U}]\,\varepsilon\,\{Q\}) &= \{[\ P \to \mathcal{U}(Q)\ ]\}\\
\mathsf{VCGu}(\{P\}[\mathcal{U}]\,\textbf{while}\ b\ \textbf{do}\,\{I\}\,W_t\,;\,W\,\{Q\}) &= \{[\ P \to \mathcal{U}(I)\ ]\}\ \cup\ \mathsf{VCGu}(\{I\ \&\&\ b\}[\emptyset]\,W_t\,\{I\})\ \cup\\
&\quad\ \mathsf{VCGu}(\{I\ \&\&\ !\,b\}[\emptyset]\,W\,\{Q\})\\
\mathsf{VCGu}(\{P\}[\mathcal{U}]\,\textbf{if}\ b\ \textbf{then}\ W_t\ \textbf{else}\ W_f\,;\,W\,\{Q\}) &= \mathsf{VCGu}(\{P\ \&\&\ b\}[\mathcal{U}]\,W_t@W\,\{Q\})\ \cup\\
&\quad\ \mathsf{VCGu}(\{P\ \&\&\ !\,b\}[\mathcal{U}]\,W_f@W\,\{Q\})
\end{aligned}$$

Fig. 16. A VCGen based on updates: VCGu

So, it follows directly from propositions 7 and 8 that to prove the validity of the triple $\{P\}\,C\,\{Q\}$ it is enough to show that $\vdash_{\mathsf{Hu}} \{P\}[\emptyset]\,\mathcal{T}(C)\,\{Q\}$.

Applying the rules of system Hu backwards to construct a proof tree is in fact very close to performing a symbolic execution of the program, since at each step of the proof, corresponding to a program $C\,;\,W$, the proof construction will proceed with the execution of the program W.

For this reason it has been argued that tools based on such a system with updates may be more adequate for debugging purposes, since the construction of proofs follows the symbolic execution of the code.

Figure 16 contains the definition of a VCGen obtained from system Hu. We remark that whereas in the case of system Hgi producing a VCGen required imposing a specific strategy for the construction of proof trees, in system Hu the construction of derivations is deterministic. It is immediate to see that the verification conditions produced by this algorithm, for a given Hoare triple with updates, are exactly the side conditions of its derivation tree.

Lemma 13 $\mathsf{VCGu}(\{P\}[\mathcal{U}]\,W\,\{Q\}) \Vdash_{\mathsf{Hu}} \{P\}[\mathcal{U}]\,W\,\{Q\}$

Proof. By induction on the structure of W.

Example 3 *We exemplify the use of this VCGen with our running example of Figure 11. Let* $P = n > 0$, $Q = x == \mathrm{Fib}(n)$, *and* $I = i \leq n\ \&\&\ x == \mathrm{Fib}(i)\ \&\&\ y == \mathrm{Fib}(i-1)$. *We start with an empty update. The set of generated verification condi-*

tions consists of the following

$$
\begin{array}{l}
\mathsf{VCGu}(\{P\}[\emptyset]\, C^{\mathrm{I}}_{\mathrm{Fib}} \,;\, \varepsilon\,\{Q\}) \\
= \mathsf{VCGu}(\{P\}[\{x \mapsto 1, y \mapsto 0, i \mapsto 1\}]\, \textbf{while}\ i < n\ \textbf{do}\, \{I\}\, C'' \,;\, \varepsilon\,\{Q\}) \\
= \{[\ P \to \{x \mapsto 1, y \mapsto 0, i \mapsto 1\}(I)\ ]\} \cup \mathsf{VCGu}(\{I\ \&\&\, i < n\}[\emptyset]\, C''\, \{I\}) \cup \mathsf{VCGu}(\{I\ \&\&\ !\,(i < n)\}[\emptyset]\, \varepsilon\,\{Q\}) \\
= \{[\ n > 0 \to 1 \leq n\ \&\&\, 1 == \mathrm{Fib}(1)\ \&\&\, 0 == \mathrm{Fib}(1-1)\ ]\}\ \cup\ \{[\ I\ \&\&\ !\,(i < n) \to \emptyset(Q)\ ]\} \cup \\
\quad \mathsf{VCGu}(\{I\ \&\&\, i < n\}[\emptyset]\, aux := y\,;\, y := x\,;\, x := x + aux\,;\, i := i+1\,;\, \varepsilon\,\{I\}) \\
= \{[\ n > 0 \to 1 \leq n\ \&\&\, 1 == \mathrm{Fib}(1)\ \&\&\, 0 == \mathrm{Fib}(1-1)\ ], \\
\qquad [\ i \leq n\ \&\&\, x == \mathrm{Fib}(i)\ \&\&\, y == \mathrm{Fib}(i-1)\ \&\&\ !\,(i < n) \to Q\ ], \\
\qquad [\ i \leq n\ \&\&\, x == \mathrm{Fib}(i)\ \&\&\, y == \mathrm{Fib}(i-1)\ \&\&\, i < n \to \\
\qquad\qquad\qquad i+1 \leq n\ \&\&\, x + y == \mathrm{Fib}(i+1)\ \&\&\, x == \mathrm{Fib}((i+1)-1)\ ]\}
\end{array}
$$

since,

$$
\begin{array}{l}
\mathsf{VCGu}(\{I\ \&\&\, i < n\}[\emptyset]\, aux := y\,;\, y := x\,;\, x := x + aux\,;\, i := i+1\,;\, \varepsilon\,\{I\}) \\
= \mathsf{VCGu}(\{I\ \&\&\, i < n\}[\emptyset\,;\, aux := y]\, y := x\,;\, x := x + aux\,;\, i := i+1\,;\, \varepsilon\,\{I\}) \\
= \mathsf{VCGu}(\{I\ \&\&\, i < n\}[\{aux \mapsto y\}\,;\, y := x]\, x := x + aux\,;\, i := i+1\,;\, \varepsilon\,\{I\}) \\
= \mathsf{VCGu}(\{I\ \&\&\, i < n\}[\{aux \mapsto y, y \mapsto x\}\,;\, x := x + aux]\, i := i+1\,;\, \varepsilon\,\{I\}) \\
= \mathsf{VCGu}(\{I\ \&\&\, i < n\}[\{aux \mapsto y, y \mapsto x, x \mapsto x + y\}\,;\, i := i+1]\, \varepsilon\,\{I\}) \\
= \{[\ I\ \&\&\, i < n \to \{aux \mapsto y, y \mapsto x, x \mapsto x + y, i \mapsto i+1\}(I)\ ]\} \\
= \{[\ i \leq n\ \&\&\, x == \mathrm{Fib}(i)\ \&\&\, y == \mathrm{Fib}(i-1)\ \&\&\, i < n \to \\
\qquad\qquad\qquad i+1 \leq n\ \&\&\, x + y == \mathrm{Fib}(i+1)\ \&\&\, x == \mathrm{Fib}((i+1)-1)\ ]\}
\end{array}
$$

Hoare logic with updates was introduced in [24] with the accompanying proof tool KeY-Hoare. The tool will read in specifications and allow the user to construct proof trees to prove them (no explicit VCGen is used). The tool also includes a first order theory.

KeY-Hoare is in fact a side-product of a much bigger effort. The full-fledged KeY tool [1] is a verification tool for Java Card programs (compatible with JML annotations) capable of handling many aspects of real-world object-oriented programs. KeY is based on JavaCardDL [9], a version of Dynamic logic [25] suited for this language. KeY also includes a symbolic debugger module.

Updates are a key ingredient of JavaCardDL; the other main ingredient, which stands at the heart of Dynamic logic, is the existence of *modalities* in the assertion language; in particular, for each program W in the programming language and assertion Q, there exists a new assertion $[W]\,Q$, interpreted informally as “if execution of W terminates, then Q will hold in the final state”.

Formulas of JavaCardDL are of the form $([W]\,Q)^{\mathcal{U}}$. This formula is interpreted as *true* in a state s if Q holds in the state that results from executing W in the state $s_{\mathcal{U}}$, if execution terminates.

The Hoare triple $\{P\}\,C\,\{Q\}$ can be written as the formula $P \to [C]\,Q$, and it is not difficult to see that system Hu can be rewritten in the syntax of Dynamic logic. Note however that, since assertions of Dynamic logic may contain programs, it may not be possible to interpret an arbitrary formula like $P \to [C]\,Q$ directly as a Hoare triple.

10 Language Extensions

In this section we discuss several extensions to the initial language. Our aim here is to provide some insight into how each of these aspects is treated by common program verification tools for real-world languages. A brief review of these tools is then given in the next and final section.

10.1 Exceptions

Adding exceptions to our language is useful not only because an exception mechanism is useful in itself, but also because it provides a means to model control-transfer commands (like **break** in C and Java).

We extend the syntax with two new commands, and a new form of specifications for exceptional termination.

$$
\begin{array}{r}
\textbf{IComm} \ni C ::= \ldots\ \mid\ \textbf{try}\, C\, \textbf{catch}\, C \mid \textbf{throw} \\
\textbf{Spec} \in S ::= \{A\}\, C\, \{A\}\ \mid\ \{A\}\, C\, \{\!|A|\!\}
\end{array}
$$

The **throw** command raises an exception (execution terminates abruptly). The command $\textbf{try}\, C\, \textbf{catch}\, C_c$ executes C and catches a possible exception raised by it, in which case the code C_c is executed.

The informal meaning of a specification $\{P\}\, C\, \{\!|Q|\!\}$ is that if the program C is executed in an initial state in which the precondition P is *true*, then either execution of C does not terminate or it terminates *with an exception raised*, in which case the postcondition Q is *true* in the final state. If the triple $\{P\}\, C\, \{Q\}$ is valid and C terminates, then it does so normally, with no exception raised.

The behaviour of **throw** and $\textbf{try}\, C\, \textbf{catch}\, C_c$ can be described axiomatically by rules like the following, where in the latter case one rule is given for normal termination of C and another for exceptional termination.

$$\frac{}{\{P\}\, \textbf{throw}\, \{\!|Q|\!\}} \quad \text{if } \models P \rightarrow Q$$

$$\frac{\{P\}\, C\, \{Q\}}{\{P\}\, \textbf{try}\, C\, \textbf{catch}\, C_c\, \{Q\}} \qquad \frac{\{P\}\, C\, \{\!|Q|\!\} \qquad \{Q\}\, C_c\, \{R\}}{\{P\}\, \textbf{try}\, C\, \textbf{catch}\, C_c\, \{R\}}$$

This presentation is a bit tedious: one would also need a second version of the last rule above, for the case where execution of C_c raises an exception. Moreover, simply adding these rules to the system Hgi would not work: one would have to add new rules to account for exceptional termination. It is simpler to use the notation $\{P\}\, C\, \{Q\}\{\!|R|\!\}$ for a specification with two possible outcomes. The intended meaning is that $\{P\}\, C\, \{Q\}\{\!|R|\!\}$ is valid if either $\{P\}\, C\, \{Q\}$ is valid or $\{P\}\, C\, \{\!|R|\!\}$ is valid.

With this notion of specification it is possible to write a goal-directed Hoare logic system for the extended language by simply adding an exceptional postcondition $\{\!|R|\!\}$ to all the Hoare triples in the system Hgi and extending it with the rules

$$\frac{}{\{P\}\, \textbf{throw}\, \{Q\}\{\!|R|\!\}} \quad \text{if } \models P \rightarrow R \qquad \frac{\{P\}\, C\, \{Q\}\{\!|R|\!\} \qquad \{R\}\, C_c\, \{Q\}\{\!|S|\!\}}{\{P\}\, \textbf{try}\, C\, \textbf{catch}\, C_c\, \{Q\}\{\!|S|\!\}}$$

The definitions of prec and VCG (Figures 9 and 13) can be easily adapted; it suffices to modify the signatures of functions to take two postconditions as arguments, including a second postcondition R in every invocation $\mathsf{prec}\,(\cdot, \cdot, R)$ and $\mathsf{VC_{aux}}\,(\cdot, \cdot, R)$. The following additional clauses are required.

$$\begin{aligned}
\mathsf{prec}\,(\textbf{throw}, Q, R) &= R \\
\mathsf{prec}\,(\textbf{try}\, C\, \textbf{catch}\, C_c, Q, R) &= \mathsf{prec}\,(C, Q, \mathsf{prec}\,(C_c, Q, R))
\end{aligned}$$

$$\begin{aligned}
\mathsf{VC_{aux}}\,(\textbf{throw}, Q, R) &= \emptyset \\
\mathsf{VC_{aux}}\,(\textbf{try}\, C\, \textbf{catch}\, C_c, Q, R) &= \mathsf{VC_{aux}}\,(C, Q, \mathsf{prec}\,(C_c, Q, R)) \cup \mathsf{VC_{aux}}\,(C_c, Q, R)
\end{aligned}$$

And finally

$$\mathsf{VCG}(\{P\}\, C\, \{Q\}\{\!|R|\!\}) = [\; P \rightarrow \mathsf{prec}\,(C, Q, R)\;] \cup\; \mathsf{VC_{aux}}\,(C, Q, R)$$

This treatment of exceptions is similar to that proposed in the work of Leino and colleagues on weakest preconditions of guarded command languages [41].

10.2 Arrays and Pointers: Aliasing

Aliasing is a phenomenon that occurs in programming languages when some form of indirection is possible in the data types.

Arrays are one such example: if we introduce arrays of integers in our language (with notation $u[k]$ for the value stored in position k of array u), then of course indexing by arbitrary expressions must be allowed – restricting indexes to be constants makes them useless collections of variables, since it is not possible to iterate over them. This creates an opportunity for aliasing to occur. In particular, the expressions $u[e]$ and $u[e']$ may refer to the same positions in the array or not, depending on whether $e = e'$ holds. This is usually called *subscript aliasing*.

A similar phenomenon occurs in a language with structures (or objects). Let $s.a$ denote the value stored in the field a of structure s, then $p.a$ and $q.a$ may refer to the same value or not, depending on whether p and q refer to the same structure. Note that this only makes sense if p and q are references (or pointers) to structures, since it is not possible for two ordinary variables to have as values the same structure. For this reason this form of aliasing is known as *pointer aliasing*.

Simply viewing arrays as indexed variables will not work. Consider the following adaptation of the weakest precondition calculation for an assignment instruction.

$$\mathsf{prec}(u[i] := e, Q) = Q[u[i] \mapsto e]$$

This would yield for instance

$$\mathsf{prec}(u[i] := 10, u[j] > 100) = u[j] > 100$$

but clearly the precondition $u[j] > 100$ will not be preserved by the command, if executed in a state in which $i = j$.

Hoare's solution to this problem was to see arrays as monolithic objects, and to have an array update operation. This approach is not so useful in the context of the generation of verification conditions. The correct output would be a logical condition that incorporates all the required comparisons between all index expressions present in the postcondition and the array position assigned to. For the above example one would have

$$\mathsf{prec}(u[i] := 10, u[j] > 100) = (i == j \rightarrow 10 > 100)\ \&\&\ (i\,!\!= j \rightarrow u[j] > 100)$$

It is relatively simple to devise an algorithm to produce this verification condition, which we leave to the reader.

The logic with updates studied in section 9 copes very easily with aliasing: it suffices to adapt to array positions or structure fields the definition of the operation that sets the value of a variable. For instance for array positions one could write

$$(\mathcal{U}\,; a[j] := e) = \mathcal{U} \oplus \{a[j] \mapsto \mathcal{U}(e)\}$$

Dealing with pointer aliasing is crucial if one wants to be able to verify programs that use recursive data-structures, dynamically defined in heap memory. A heap can be seen as a very big array (indexed by memory positions), and in this sense pointer aliasing is an instance of index aliasing. Seriously reasoning about pointer programs requires a new framework that avoids the proliferation of arithmetic proof obligations concerning indexes. The theoretical advances in this area have been developed in the context of Separation logic [47], whose key idea is the possibility to explicitly express the separation between different structures.

An alternative to this approach had previously been proposed by Burstall [13] and further explored by Bornat [12]. The idea here was to see the heap as a *collection* of arrays, rather than a single array. In particular, there should be one such array for every structure/object field. For instance, $p.a$ and $q.a$ could be represented by positions in the same (heap model) array a. They will be represented by the same position if p and q are the same memory address (thus the same index of a). This approach has the advantage of producing first-order verification conditions, which means that it can be included in a standard VCGen.

The Caduceus VCGen uses this approach to construct a memory model for C programs. The heap model is explicitly constructed as a set of data-structures in the language of the Why generic VCGen (see Section 11 below).

10.3 Procedure Calls

Procedures interact with auxiliary variables in a way that destroys completeness of a Hoare-style logic for the extended language. The theoretical problems raised by introducing (mutually-recursive) procedures and the solutions proposed for them (typically involving modified structural rules and a view of the inference system oriented to judgments, with Hoare triples possibly assumed valid in the context) are outside the scope of this paper. The reader is referred to [2, 35, 51] for details.

It is easy to see that recursive procedures raise a difficulty similar to the one that leads to the introduction of loop invariants. Suppose procedure **f** has as body the code $C_{\mathbf{f}}$; then the following straightforward rule transfers the proof of correctness to the procedure body.

$$\frac{\{P\}\, C_{\mathbf{f}}\, \{Q\}}{\{P\}\, \mathbf{call\, f}\, \{Q\}}$$

and it is immediate to see that if $C_{\mathbf{f}}$ contains a call to **f** this may generate infinite derivations.

We outline here the approach taken by modern program verification systems, which consists in annotating procedures with a precondition and a postcondition, i.e. to force the inclusion of specifications at the level of procedures. Homeier and Martin's mechanically verified VCGen has been extended to a language with recursive procedures (with parameters) following this approach [28].

We assume a set of procedure names $\mathcal{F}$ and let **f** range over $\mathcal{F}$. We let **Proc** denote the class of procedure definitions (the body of a procedure definition is really just a specification). We extend the command syntax with procedure calls, and let programs be defined as sequences of procedure definitions.

$$\begin{aligned} \mathbf{IComm} \ni C &::= \ldots \mid \mathbf{call\, f} \\ \mathbf{Proc} \ni \Phi &::= \mathbf{proc\ f} = \{A\}\, C\, \{A\} \\ \mathbf{Prog} \ni \Pi &::= \Phi\, \Pi \mid \Phi \end{aligned}$$

We say that a program is *well-defined* if the names of its procedures are pairwise disjoint.

One particular procedure should be designated as an entry point into each program, but we will abstract away from this operational issue. From the axiomatic point of view, all procedures should meet their specifications, which means that the generated set of verification conditions will be the union of the sets generated for each individual procedure.

Apart from that, there is the **call** command to consider. The specification of each procedure establishes a sort of 'contract' between the calling procedure and the invoked procedure: if $\mathbf{f}_1$ calls $\mathbf{f}_2$ then $\mathbf{f}_1$ should ensure that the precondition of $\mathbf{f}_2$ is satisfied immediately before the call; the specification of $\mathbf{f}_2$ guarantees that its postcondition will be satisfied when control is returned to $\mathbf{f}_1$. This principle is thoroughly explored in the so-called *design-by-contract* approach to software development.

Let $\mathbf{proc\ f} = \{P_{\mathbf{f}}\}\, C_{\mathbf{f}}\, \{Q_{\mathbf{f}}\} \in \Pi$ with Π a well-defined program. The following rule is all that is necessary to complement the system Hgi.

$$\frac{}{\{P\}\, \mathbf{call\, f}\, \{Q\}} \quad \text{if } \models P \to P_{\mathbf{f}} \text{ and } \models Q_{\mathbf{f}} \to Q$$

Note that the rule enjoys the sub-formula property, since $P_{\mathbf{f}}$ and $Q_{\mathbf{f}}$ are obtained from **f**.

The following clauses complement accordingly the definition of the auxiliary functions of VCG, which now become relative to a given program Π.

$$\begin{aligned} \mathsf{prec}^{\Pi}(\mathbf{call\, f}, Q) &= P_{\mathbf{f}} \\ \mathsf{VC}^{\Pi}_{\mathsf{aux}}(\mathbf{call\, f}, Q) &= \{[\ Q_{\mathbf{f}} \to Q\ ]\} \end{aligned}$$

The verification conditions for the whole program guarantee that each procedure keeps to its part of the contract, when invoked. They are given by

$$\mathsf{VCG}(\Pi) = \bigcup_{\Phi \in \Pi} \mathsf{VCG}(\Phi)$$

This system is similar (although presented is a slightly different way) to that of [8]. We remark that this notion of procedure without parameters, which relies on the use of global variables, is appropriate to capture the behaviour of methods in object-oriented programming languages. In this case, global variables correspond to class or instance variables.

We briefly consider now the case of procedures with parameters. Consider the following procedure definition, where x is the only formal parameter of $\mathbf{f}$, of type **int**.

$$\mathbf{proc}\ \mathbf{f}(x) = \{P_{\mathbf{f}}\}\, C_{\mathbf{f}}\, \{Q_{\mathbf{f}}\}$$

One novelty is that the variable x is *local* to the procedure (one could also have variables that are local to a *block* of code, but in this paper we haven't considered this possibility). One first difficulty is that in general this introduces a new form of aliasing, which is present if x is also used as a *global* variable. In this case, it makes sense that occurrences of x in annotations internal to the procedure definition be interpreted as referring to the local variable. However, this makes it impossible to refer to the value of the global variable x. This form of aliasing is called *parameter aliasing*. We avoid it here by simply assuming that global and local variables are taken from disjoint alphabets.

Note also that it must be possible for the precondition and postcondition of a procedure specification to contain occurrences of the parameter variables. Following the discussion at the end of Section 7, it may be necessary to resort to auxiliary variables or an operator like old to be able to refer, in the postcondition of a procedure, to the entry value of a parameter.

When the procedure is called with an actual parameter e, using a command $\mathbf{call}\,\mathbf{f}(e)$, the contract between the invoking procedure and $\mathbf{f}$ implies that the precondition should be met, with the actual parameter substituted by the formal one. This is formalized as the following rule

$$\frac{}{\{P\}\, \mathbf{call}\, \mathbf{f}(e)\, \{Q\}} \quad \text{if} \ \models P \rightarrow P_{\mathbf{f}}[x \mapsto e] \ \text{ and } \ \models Q_{\mathbf{f}} \rightarrow Q$$

and the corresponding clause in the weakest precondition function would now be

$$\mathsf{prec}^{\Pi}(\mathbf{call}\, \mathbf{f}(e), Q) = P_{\mathbf{f}}[x \mapsto e]$$

A final remark: the treatment of call-by-reference parameters further complicates things; it requires a formalization of pointers as discussed at the end of the previous subsection.

Functions. A function is a subroutine that returns a value upon completion. Our language could be extended with functions returning integers as follows ($\mathcal{F}$ can be used also as an alphabet of function names).

$$\begin{aligned} \mathbf{IComm} \ni C &::= \ldots \mid x := \mathbf{fcall}\, \mathbf{f}(e) \\ \mathbf{Proc} \ni \Phi &::= \ldots \mid \mathbf{fun}\ \mathbf{f}(x) = \{A\}\, C\, ;\, \mathbf{return}\, e\, \{A\} \end{aligned}$$

We assume every function definition contains a single **return** statement at the end, and moreover function calls can only occur directly in assignment commands.

In order to make it possible to write useful specifications of functions, the annotation language has to be extended once again. We use the keyword 'result' in the postcondition of a function specification, to refer to the value returned by the function. 'result' is a logical variable that may only occur in postconditions of functions. For instance, one could specify a function that calculates Fibonacci numbers as

$$\mathbf{fib}(n) = \{n > 0\}\, C_{\mathbf{fib}}\, \{\text{result} == \text{Fib}(n)\}$$

The function body could be implemented as $C_{\mathbf{fib}} = (C^{\text{I}}_{\text{Fib}}\, ;\, \mathbf{return}\, x)$, where $C^{\text{I}}_{\text{Fib}}$ is the code in Figure 11.

Let $\mathbf{fun}\ \mathbf{f}(x) = \{P_{\mathbf{f}}\}\, C\, ;\, \mathbf{return}\, e\, \{Q_{\mathbf{f}}\} \in \mathit{\Pi}$. The verification conditions are calculated in the same way as before, with new rules for function calls and for the result keyword.

$$\frac{}{\{P\}\, y := \mathbf{fcall}\, \mathbf{f}(e)\, \{Q\}} \quad \text{if} \models P \rightarrow P_{\mathbf{f}}[x \mapsto e] \text{ and } \models Q_{\mathbf{f}}[\text{result} \mapsto y] \rightarrow Q$$

$$\frac{\{P\}\, C\, \{Q[\text{result} \mapsto e]\}}{\{P\}\, C\, ;\, \mathbf{return}\, e\, \{Q\}}$$

And the VCGen of Section 7 could be modified by adding the following clauses.

$$\begin{aligned} \mathsf{prec}^{\mathit{\Pi}}(y := \mathbf{fcall}\, \mathbf{f}(e), Q) &= P_{\mathbf{f}}[x \mapsto e] \\ \mathsf{VC}^{\Pi}_{\mathsf{aux}}(y := \mathbf{fcall}\, \mathbf{f}(e), Q) &= \{[\ Q_{\mathbf{f}}[\text{result} \mapsto y] \rightarrow Q\ ]\} \end{aligned}$$

$$\begin{aligned} \mathsf{prec}^{\mathit{\Pi}}(C\, ;\, \mathbf{return}\, e, Q) &= \mathsf{prec}^{\mathit{\Pi}}(C, Q[\text{result} \mapsto e]) \\ \mathsf{VC}^{\Pi}_{\mathsf{aux}}(C\, ;\, \mathbf{return}\, e, Q) &= \mathsf{VC}^{\Pi}_{\mathsf{aux}}(C, Q[\text{result} \mapsto e]) \end{aligned}$$

Notice that the variable 'result' does not occur in the verification conditions produced by the VCGen.

Frame Properties. In languages like C or Java, evaluation of expressions may have *side effects* that modify the state of the program; `x++` is a typical example of such an expression. This issue has been addressed in most attempts to construct Hoare logics for realistic languages, see Section 11.

A function call is also an example of an expression with side effects. A command like $x := \mathbf{fcall}\, \mathbf{f}(e)$ alters the state of the program in a way that may not be limited to x, since execution of the body of **f** is free to access global variables (or class and instance variables, in the case of object-oriented programs). To allow for modular reasoning, the function's contract can include information describing the unchanged part of the state, using a technique similar to that described at the end of Section 7. However, it is easy to see that this approach is inadequate. For instance, if new variables are introduced in the state, the specifications of many different functions may have to be changed. This is usually known as the *frame problem* [11], and it applies to all forms of subroutines, including procedures, functions, and object methods – it is in fact particularly important in object-oriented languages.

In general terms, it is more useful to include in contracts a description of what part of the state the subroutines are allowed to affect, rather than what part remains unchanged. This is usually known as the subroutine's *frame condition.* For instance in JML there are two frame annotations that may be included in specifications: `assignable` annotations, which list the variables that may be assigned by a method, and `modifies` annotations, which list variables whose value is allowed to be modified. Thus a variable that may be assigned inside a method, but whose value upon exit must be guaranteed to be restored to its entry value can be listed as assignable but not as modifiable.

Frame conditions are part of a method's contract: typically they give rise to proof obligations when the method is being verified, and generate hypotheses that may be used when reasoning about calls to that method. In JML, methods that do not affect the state at all are called *pure.* A method annotated as pure can be used for specification purposes and called inside annotations.

11 Conclusion: Realistic Programming Languages

The gap between the point where this paper stops, and program verification for real-world languages begins, has been the object of significant work in recent years. We finish the paper with a brief survey of how this work is being carried out.

One first aspect is that much of the recent work on program verification has been developed around a community federated around the Java Modeling Language (JML) [38, 36]. This is an annotation language for Java programs with support for preconditions, postconditions, frame conditions, and loop invariants, all written using the same syntax for expressions as the source language (exclusive of side-effects). JML also includes special keywords `\old` and `\result` with a similar interpretation to what has been explained in Sections 7 and 10.3.

The language incorporates many other modeling aspects that are useful to reason about object-oriented programs, such as class invariants (properties on the values of instance and class variables that must be preserved by all methods) and specification-only (or *model*) fields and methods, which allow for reasoning about hidden (or not yet implemented) specifications. It also includes a rich library of types with accompanying mathematical theories, including sets and sequences.

JML was not developed exclusively with program verification or other static checking methods in mind. In fact, the design-by-contract approach to software development is supported by a number of different tools for different tasks. These include dynamic assertion checkers, unit test class generators, and even generators that can be used to help write specifications. A number of program verification systems discussed below also use JML as specification language.

Admittedly, the material in Section 10 barely starts to cover the complex language constructs and memory models found in programming languages like C, Java or C#. For the particular case of object-oriented languages, there are many specific aspects that we have not covered at all. Poetzsch-Heffter and Müller [45] summarise as follows the difficulties involved in designing Hoare-style logics for these languages.

> *Three aspects make verification of OO-programs more complex than verification of programs with just recursive procedures and arbitrary pointer data structures: Subtyping, abstract types, and dynamic binding. Subtyping allows variables to hold objects of different types. Abstract types have different or incomplete implementations. Thus, techniques are needed to formulate type properties without referring to implementations. Dynamic binding destroys the static connection between the calls and the body of a procedure.*

A discussion of how these issues are addressed is beyond the scope of this paper; in the following we will concentrate instead on the organisational aspects. Accounts of the state of the art and current challenges in the field of object-oriented program verification (and specification languages) have been given by Jacobs, Kiniry and Warnier [31] and more recently by Leavens, Leino, and Müller [37].

The work on verification conditions for realistic languages falls roughly into two categories. Some researchers have proposed logics that attempt to fully describe the axiomatic semantics of the programming language; this is the case for instance of the following, which have been used to produce working program verification systems.

– Poetzsch-Heffter and Müller's logic for sequential Java [45], used in the Jive [43] verification platform to generate verification conditions exported to the PVS theorem prover. Following Gordon and Homeier and Martin, both the operational semantics of the language and the proposed Hoare inference system have been encoded in Higher-order logic, and the latter proved sound with respect to the former.

– In the context of the Loop project, a Hoare logic for JML has been developed [32]. The approach followed in Loop is quite different from what has been discussed in this paper: a compiler first produces a logical theory (in the language of the PVS theorem prover) that describes the behaviour of a given JML-annotated Java program; program correctness is then proved in this theory. However, a weakest-precondition function can also be integrated in the tool [30], with the aim of reducing the required user interaction. This function is defined inside the logic, and works on translated programs.
– The KeY project's JavaCardDL [9] is used in an integrated tool that supports the development of JML-annotated JAVA CARD programs. JavaCardDL has not to our knowledge been mechanically proved correct with respect to a non-axiomatic semantics. This system uses a special-purpose theorem prover to construct proofs of JavaCardDL.

The architectures of the latter two systems do not use a separate VCGen component, and do not export proof obligations free of program constructs.

Von Oheimb's Hoare logic for a subset of JAVA CARD [52], although not used (to our knowledge) to produce a working tool, was an important development since it was mechanically proved sound *and complete* with respect to the operational semantics, using the Isabelle/HOL interactive theorem prover.

A second approach has concentrated on developing practical tools for program verification. The focus here has not been so much on correctness – these tools do not promise to find all errors a program may contain – but on usability, integration into the software development process, and compatibility with different proof tools.

We mention here two VCGen tools that are based on relatively small and simple *intermediate languages*, sufficiently expressive to support the translation of realistic languages. The more complex aspects are handled by an adequate translation into the intermediate language, rather than by the design of a complex Hoare logic for the target language. A good example of this is the memory model: the intermediate language will typically have a very simple model with no heap, and the translation of the target language encodes the heap in some way into that model, to ensure an adequate treatment of references/pointers and aliasing.

The Boogie [5] program verifier was designed for the Spec# language (which stands for C# as JML stands for Java) and incorporates many advanced features, including integration with a development environment and automatic inference of loop invariants. The BoogiePL intermediate language is a guarded command language that features procedures and mutable variables, but excludes more complex features like methods, side-effects, or call-by-reference. So the translation of the target language into this intermediate language is responsible for encoding these features in a sound way.

The Why tool [19] was born from a very different perspective: the basic idea is to provide an interpretation of programs in type theory, such that the proof obligations for typability of the interpretation function coincide with the verification conditions. In this way, interpreting programs in type theory becomes an alternative to using a standard VCGen. The Why intermediate language is ML-like, including both functional and imperative features, exceptions, and labelling of statements (to make the use of auxiliary variables unnecessary). The type system accounts for *effects*, i.e. the type of an expression is annotated with, for instance, the set of variables that are possibly modified by its evaluation, which makes possible to eliminate aliasing. Why supports a variety of theorem provers, and has been used to produce program verification tools for both C (Caduceus [18]) and Java (Krakatoa [42]).

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The key tool. *Software and System Modeling*, 4(1):32–54, 2005.
2. Krzysztof R. Apt. Ten years of hoare's logic: A survey - part 1. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.
3. A. Azurat and I.S.W.B. Prasetya. A survey on embedding programming logics in a theorem prover. Technical report, University of Utrecht, 2002.
4. Roland Backhouse. *Program Construction – Calculating Implementations from Specifications.* John Wiley & Sons, Ltd, 2003.
5. 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 Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.
6. Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT Softw. Eng. Notes*, 31(1):82–87, 2006.
7. Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114. IEEE Computer Society, 2004.
8. Gilles Barthe, Tamara Rezk, and Ando Saabas. Proof obligations preserving compilation. In Theodosis Dimitrakos, Fabio Martinelli, Peter Y. A. Ryan, and Steve A. Schneider, editors, *Formal Aspects in Security and Trust*, volume 3866 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2005.
9. Bernhard Beckert. A dynamic logic for the formal verification of java card programs. In Isabelle Attali and Thomas P. Jensen, editors, *Java Card Workshop*, volume 2041 of *Lecture Notes in Computer Science*, pages 6–24. Springer, 2000.
10. Yves Bertot. Theorem proving support in programming language semantics. *CoRR*, abs/0707.0926, 2007.
11. Alexander Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *Software Engineering*, 21(10):785–798, 1995.
12. Richard Bornat. Proving pointer programs in hoare logic. In Roland Carl Backhouse and José Nuno Oliveira, editors, *MPC*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer, 2000.
13. Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7, 1972.
14. David R. Cok and Joseph Kiniry. Esc/java2: Uniting esc/java and jml. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.
15. Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *SPC*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer, 2005.
16. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
17. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
18. Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of c programs. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.
19. Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.
20. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 193–205, New York, NY, USA, 2001. ACM.

21. Donald I. Good. Mechanical proofs about computer programs. Technical Report 41, The University of Texas at Austin, March 1984.
22. Michael J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current trends in hardware verification and automated theorem proving*, pages 387–439. Springer-Verlag New York, Inc., 1989.
23. David Gries. *The Science of Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1987.
24. Reiner Hähnle and Richard Bubel. A hoare-style caulculus with explicit state updates. Department of Computer Science, Chalmers University of Technology.
25. D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic Volume II — Extensions of Classical Logic*, pages 497–604. D. Reidel Publishing Company: Dordrecht, The Netherlands, 1984.
26. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
27. Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *Comput. J.*, 38(2):131–141, 1995.
28. Peter V. Homeier and David F. Martin. Mechanical verification of mutually recursive procedures. In Michael A. McRobbie and John K. Slaney, editors, *CADE*, volume 1104 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 1996.
29. Shigeru Igarashi, Ralph L. London, and David C. Luckham. Automatic program verification i: A logical basis and its implementation. *Acta Inf.*, 4:145–182, 1974.
30. Bart Jacobs. Weakest pre-condition reasoning for java programs with jml annotations. *J. Log. Algebr. Program.*, 58(1-2):61–88, 2004.
31. Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java program verification challenges. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 2852 of *Lecture Notes in Computer Science*, pages 202–219. Springer, 2002.
32. Bart Jacobs and Erik Poll. A logic for the java modeling language jml. In Heinrich Hußmann, editor, *FASE*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2001.
33. Cliff B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Ann. Hist. Comput.*, 25(2):26–49, 2003.
34. Anne Kaldewaij. *Programming: the derivation of algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.
35. Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.
36. G. Leavens and Y. Cheon. Design by contract with jml, 2003.
37. Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, 2007.
38. Gary T. Leavens, Clyde Ruby, K. Rustan, M. Leino, Erik Poll, and Bart Jacobs. Jml: notations and tools supporting detailed design in java. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 105–106, New York, NY, USA, 2000. ACM.
39. K. Rustan M. Leino. Extended static checking: A ten-year perspective. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 157–175, London, UK, 2001. Springer-Verlag.
40. K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.
41. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking java programs via guarded commands. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 110–111, London, UK, 1999. Springer-Verlag.
42. Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The krakatoa tool for certificationof java/javacard programs annotated in jml. *J. Log. Algebr. Program.*, 58(1-2):89–106, 2004.
43. J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. FernUniversität Hagen, 2000.
44. G. C. Necula. Proof-carrying code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.

45. Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential java. In S. Doaitse Swierstra, editor, *ESOP*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.
46. John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, England, 1998.
47. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
48. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.
49. T. Schreiber. Auxiliary variables and recursive procedures, 1997.
50. R. D. Tennent. *Specifying Software*. Cambridge University Press, New York, NY, USA, 2001.
51. David von Oheimb. Hoare logic for mutual recursion and local variables. In *Foundations of Software Technology and Theoretical Computer Science*, pages 168–180, 1999.
52. David von Oheimb. Hoare logic for java in isabelle/hol. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.
53. Glynn Winskel. *The Formal Semantics of Programming Languages: An introduction*. Foundations of Computing. The MIT Press, Cambridge, Massachusetts, 1993.