Verification Graphs for Programs with Contracts

Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto

Techn. Report TR-HASLab:01:2012

February 2012

HASLab - High-Assurance Software Laboratory
Universidade do Minho
Campus de Gualtar – Braga – Portugal
http://haslab.di.uminho.pt

TR-HASLab:01:2012
Verification Graphs for Programs with Contracts
by Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto

Abstract

The *Design by Contract* (DbC) approach to software development enables the formal verification of programs, or program components, as well as their safe reuse. Specification languages for common programming languages, namely the Java Modeling Language (JML) for Java, and Spec# for C#, have in this context become very popular.

This paper establishes a basis for the generation of verification conditions combining forward and backward reasoning, for programs consisting of mutually-recursive procedures annotated with contracts and loop invariants. There exist in general many different sets of verification conditions that can be obtained in this way, and we give a characterization of these sets based on edge labelings of the control flow graphs of the procedures.

We also show how our approach can cope with the efficient techniques for generating verification conditions that have been proposed for *passive programs*. Finally, the paper discusses applications of the approach, including error path discovery and interactive verification.

1 Introduction

A central issue in program verification is the generation of *verification conditions* (VCs): proof obligations which, if successfully discharged, guarantee the correctness of a program vis-à-vis a given specification. While the basic theory of program verification has been around since the 1960s, the late 1990s saw the advent of practical tools for the verification of realistic programs, and research in this area has been very active since then. Automated theorem provers have contributed decisively to these developments.

There are two well-established methods for producing sets of verification conditions, based respectively on *backward propagation* and *forward propagation* of assertions. Verifying the behavior of programs is of course in general an undecidable problem, and as such automation is necessarily limited. Typical tools require the user to provide additional information, in particular *loop invariants*, and one could be tempted to think that the only interesting problem of program verification is automating the generation of invariants: if appropriate annotations are provided, then generating and proving verification conditions should be straightforward. It would however be wrong to think that the method used for generating verification conditions is not important. It can be important for the following reasons:

- The choice of employing backward propagation or forward propagation of assertions (which as we shall see are close to weakest precondition and strongest postcondition calculations respectively) is not indifferent. Program verification tools typically resort to the former, since the latter requires the introduction of existential quantifiers in the generated VCs. The use of strongest postcondition calculations has however been gaining momentum in a series of recent papers (see below).
- It has been shown that there are serious efficiency issues involved: a naive algorithm can produce verification conditions of exponential size on the length of the program, thus compromising any hope of automating the verification process [12]. Fortunately, it is now well understood how to prevent this exponential explosion (see Section 5). It has also been understood that splitting VCs into smaller formulas may help the performance of theorem provers (see Section 6). Automatic invariant generation techniques [6, 7] also tend to produce very large invariants, which is an additional factor affecting efficiency.
- The above two issues have to do with the "quality" of the sets of VCs generated, with impact on the feasibility of the automated proofs. Other aspects have to do with the interpretation of invalid verification conditions: it is important to be able to identify the particular execution paths that cause them. A method for producing VCs that facilitates this is advisable for debugging applications.
- Finally, let us recall that a failed verification may be due not only to an incorrect program, but also possibly to annotations that are not compatible with the program. If the 'invariant' annotation provided by the user for a given loop is not in fact an invariant, then the program cannot be proved correct. An *interactive* approach to VC generation can help the users find the appropriate annotations for their programs.

The Case for Forward Propagation

The forward propagation of assertions was a key part of Floyd's inductive assertion method for reasoning about programs [13]. Strongest postcondition calculations also propagate assertions forward, and have been repeatedly advocated as a tool to assist in different software engineering tasks [27, 15]. Mike Gordon [17] has more recently argued for the advantages of forward reasoning, pointing out the similarities between strongest postcondition calculations and *symbolic execution*, a method for analyzing and checking properties of programs based on simulating their execution with symbolic input values [28]. An account of verification conditions based on forward propagation provides an interesting link with techniques based on symbolic execution.

A more recent work [22] presents another argument for the use of forward propagation: in some situations it allows for the verification conditions to be simplified while they are computed, using standard optimizations like constant propagation and even dead code elimination. Consider for instance the program $x := 10; y := 5 * x$. Its VCs can be calculated using instead the program $x := 10 \,;\, y := 50$. And for the program $x := 10 \,;\, y := 5 * x \,;\, \mathbf{if}\ y > 0\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$, they can be calculated instead from $x := 10 \,;\, y := 50 \,;\, S_1$. The resulting VCs are significantly simpler.

The resurgent use of strongest postconditions in a number of recent papers is also due to the fact that it is now known how to compute them in a way that is uncluttered by existential quantifiers, and moreover produces VCs that are at least as small as those generated for the same program using backward propagation (and arguably smaller, if simplifications such as mentioned above are carried out). We will elaborate on this in Section 5.

This Paper

The main goal of the present paper is to investigate how forward propagation and backward propagation methods can be combined to generate verification conditions, and how this can be put to use in a number of ways, in particular for splitting VCs and identifying error paths.

1. We show how every subblock of a given program can be analyzed by propagating an assertion forward through a prefix of the subblock, and propagating another assertion backward through the complementary suffix. It is straightforward to generate a set of VCs for a block if we stipulate that all subblocks before (resp. after) a given statement are analyzed using forward (resp. backward) propagation. Loops complicate this simple setting in a way that will be explained in detail.
2. Fixing the location of the border statement between the prefix and the suffix does not result in a unique set of VCs, since each subblock of S (located either before or after that statement) can in turn be analyzed using both forms of propagation. We introduce a precise characterization of all the sets of verification conditions for a program block, based on labelings of the control flow graph of the block.
3. We show that the above results are compatible with the recently proposed efficient methods for computing VCs of passive programs. In fact for this class of programs it becomes clear that the different sets of verification conditions contain formulas that are syntactically very close, and for this reason the choice of strategy for generating VCs becomes less important.
4. The previous point seems to imply that being able to combine backward and forward propagation is not so useful for improving the automated verification of passive programs. However, we show that our approach has other applications, including the user-guided generation of verification conditions and the definition of alternative verification strategies, for instance for the identification of error paths or for VC splitting. We briefly describe a prototype tool that implements a visual interactive VC generator based on the above principles.

Context: the GamaAnimator Project

The general framework to which we adhere in this paper is the verification of programs based on their contracts. The *Design by Contract* (DbC) approach to software development [26] facilitates modular verification and certified code reuse. The contract of a component (or procedure, or method) can be regarded as a form of enriched software documentation that specifies the behaviour of that component. The development and broad adoption of annotation languages for the most popular programming languages reinforces the importance of using DbC principles in the development of programs. These include for instance the Java Modeling Language (JML) [9]; Spec# [4], a formal language for C# contracts; and the SPARK [1] subset of Ada.

Annotating programs to formalize contracts is not a trivial task for most software engineers, who are used to coding their algorithms in traditional programming languages. Creating tools that provide a clear view of how annotations such as preconditions, postconditions, and invariants are propagated through the code is an aim that we pursue in the GamaSlicer project, which is the context for the work described here. In particular this work has stemmed from our research on *slicing* programs based on specifications [5].

Organization of the Paper Section 2 introduces our setting for the verification of contract-annotated programs (sets of mutually recursive procedures), and shows how forward propagation and backward propagation can be simultaneously used to produce sets of verification conditions. Section 3 introduces two labelings of the control flow graph of a procedure, implementing backward and forward propagation of assertions through the graph, and Section 4 investigates the sets of edge conditions obtained from these labelings by constructing implicative formulas. It is shown how (minimal) sets of verification conditions can be identified, corresponding to arbitrary combinations of forward and backward propagation of assertions. Section 5 shows how verification conditions can be greatly optimized if programs are first converted into a so-called passive form, and shows how our approach copes with these optimizations. Section 6 discusses applications of our approach, and Section 7 concludes the paper. A contains proofs, and B briefly describes our GamaAnimator prototype tool.

2 Background: Verification Conditions and Contracts

To illustrate our ideas we use a simple programming language. Its syntax is defined in Figure 1, in two levels (x and $\mathbf{p}$ range over sets of variables and procedure names respectively). First we form blocks (or sequences)

$$
\begin{array}{llll}
\mathbf{Exp[int]} & \ni & e & ::= \quad \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x \mid -e \mid e + e \mid e - e \mid e * e \mid e \,\mathsf{div}\, e \mid e \,\mathsf{mod}\, e \\
\mathbf{Exp[bool]} & \ni & b & ::= \quad \mathsf{true} \mid \mathsf{false} \mid e = e \mid e < e \mid e \leq e \mid e > e \mid e \geq e \mid e \neq e \mid \\
 & & & \quad\; b \wedge b \mid b \vee b \mid \neg b \\
\mathbf{Assert} & \ni & A & ::= \quad \mathsf{true} \mid \mathsf{false} \mid e = e \mid e < e \mid e \leq e \mid e > e \mid e \geq e \mid e \neq e \mid \\
 & & & \quad\; A \wedge A \mid A \vee A \mid \neg A \mid A \rightarrow A \mid \forall x.\, A \mid \exists x.\, A \\
\mathbf{Comm} & \ni & C & ::= \quad \mathbf{skip} \mid x := e \mid \mathbf{if}\ b\ \mathbf{then}\ S\ \mathbf{else}\ S \mid \mathbf{while}\ b\ \mathbf{do}\ \{A\}\, S \mid \mathbf{call}\ \mathbf{p} \\
\mathbf{Block} & \ni & S & ::= \quad C \mid C\,;\, S \\
\mathbf{Proc} & \ni & \Phi & ::= \quad \mathbf{pre}\ A\ \mathbf{post}\ A\ \mathbf{proc}\ \mathbf{p} = S \\
\mathbf{Prog} & \ni & \Pi & ::= \quad \Phi \mid \Pi\ \Phi
\end{array}
$$

Figure 1: Programming language syntax

of commands, which correspond to programs of a standard *While* programming language. These include a procedure call command, in addition to **skip**, assignment, conditionals, and loops. Each loop is additionally annotated with an assertion, interpreted as a *loop invariant*. The language of assertions extends boolean expressions with implication and first-order quantification. Procedures can then be defined, consisting of a block of code annotated with two assertions that form the procedure's specification, or *contract*.

A *program* is a non-empty sequence of (mutually recursive) procedure definitions (operationally, an entry point would have to be defined for each program, but that is not important for our current purpose). For the sake of simplicity we consider only *parameterless procedures* that share a set of global variables, but the ideas presented here can be adapted to cope with parameters (passed by value or by reference), as well as return values. Note that a program defined in this way is close to the notion of class in object-oriented programming, with procedures and global variables playing the role of methods and attributes. Other notions like the creation of class instances or inheritance are absent from this analogy.

A program is well-formed if the name of every procedure defined in it is unique and the program is closed with respect to procedure invocation. We will write $\mathcal{P}(\Pi)$ for the set of names of procedures defined in the program Π. The operators **pre**, **post**, and **body** will be used to refer to the two assertions that form the procedure's contract and body command respectively, i.e. given the procedure definition $\mathbf{pre}\ P\ \mathbf{post}\ Q\ \mathbf{proc}\ \mathbf{p} = S$ with $\mathbf{p} \in \mathcal{P}(\Pi)$, one has $\mathbf{pre}_\Pi(\mathbf{p}) = P$, $\mathbf{post}_\Pi(\mathbf{p}) = Q$, and $\mathbf{body}_\Pi(\mathbf{p}) = S$. The program name will be omitted when clear from context.

2.1 Intraprocedural Verification Based on Backward Propagation

Our language differs from the standard *While* language with parameterless procedures in that programs contain loop invariants as annotations. Given an (annotated) block S, let $\underline{S}$ be the *While* program that results from erasing the annotations. In what follows we will use this notation in order to formalize a relation between the set of verification conditions of an annotated block S and the partial correctness of $\underline{S}$ with respect to a specification.

Consider for now command blocks not containing procedure calls. There are two standard frameworks for studying the correctness of such a *While* program with respect to a specification. The first is the axiomatic approach, based on the use of a program logic such as Hoare logic [20]; the second approach is based on Dijkstra's predicate transformers, which requires the translation of programs into a guarded commands language [10]. In this paper we adhere to the first framework (this is not however a very important choice – our ideas could also be developed using predicate transformers).

The *Hoare triple* $\{P\}\,\underline{S}\,\{Q\}$ denotes the partial correctness of $\underline{S}$ with respect to the precondition P and the postcondition Q. The triple is valid when Q holds after execution of $\underline{S}$ terminates (if it does terminate), starting from an initial state that satisfies P. Let $(\underline{S}, s) \Downarrow s'$ denote the fact that when executed in the initial state s, $\underline{S}$ stops in the final state s' ($\Downarrow$ is the evaluation relation of a standard operational semantics for *While* programs). Then the validity of a triple is established formally by the following interpretation:

$$[\![\{P\}\,\underline{S}\,\{Q\}]\!] \quad = \quad \text{for all states } s, s',\ [\![P]\!](s) \wedge (\underline{S}, s) \Downarrow s' \Rightarrow [\![Q]\!](s')$$

We remark that throughout the paper we will use the word *specification* to refer to a pair of assertions (P, Q); we will sometimes abuse language and use the words *precondition* and *postcondition* to refer to the

$$
\begin{aligned}
\mathsf{wprec}(\mathbf{skip}, Q) &= Q \\
\mathsf{wprec}(x := e, Q) &= Q[e/x] \\
\mathsf{wprec}(\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f, Q) &= (b \to \mathsf{wprec}(S_t, Q)) \wedge (\neg b \to \mathsf{wprec}(S_f, Q)) \\
\mathsf{wprec}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S, Q) &= I \\
\mathsf{wprec}(\mathbf{call}\ \mathbf{p}, Q) &= \forall \overline{x_f}.\, (\forall \overline{y_f}.\, \mathbf{pre}(\mathbf{p})[\overline{y_f}/\overline{y}] \to \mathbf{post}(\mathbf{p})[\overline{y_f}/\overline{y}, \overline{x_f}/\overline{x}]) \to Q[\overline{x_f}/\overline{x}] \\
\mathsf{wprec}(C; S, Q) &= \mathsf{wprec}(C, \mathsf{wprec}(S, Q))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{wvc}(\mathbf{skip}, Q) &= \emptyset \\
\mathsf{wvc}(x := e, Q) &= \emptyset \\
\mathsf{wvc}(\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f, Q) &= \mathsf{wvc}(S_t, Q) \cup \mathsf{wvc}(S_f, Q) \\
\mathsf{wvc}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S, Q) &= \{I \wedge b \to \mathsf{wprec}(S, I), I \wedge \neg b \to Q\} \cup \mathsf{wvc}(S, I) \\
\mathsf{wvc}(C; S, Q) &= \mathsf{wvc}(C, \mathsf{wprec}(S, Q)) \cup \mathsf{wvc}(S, Q)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{spost}(\mathbf{skip}, P) &= P \\
\mathsf{spost}(x := e, P) &= \exists v.\, P[v/x] \wedge x = e[v/x] \\
\mathsf{spost}(\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f, P) &= \mathsf{spost}(S_t, P \wedge b) \vee \mathsf{spost}(S_f, P \wedge \neg b) \\
\mathsf{spost}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S, P) &= I \wedge \neg b \\
\mathsf{spost}(\mathbf{call}\ \mathbf{p}, P) &= \exists \overline{x_f}.\ P[\overline{x_f}/\overline{x}] \wedge (\forall \overline{y_f}.\, \mathbf{pre}(\mathbf{p})[\overline{y_f}/\overline{y}, \overline{x_f}/\overline{x}] \to \mathbf{post}(\mathbf{p})[\overline{y_f}/\overline{y}]) \\
\mathsf{spost}(C; S, P) &= \mathsf{spost}(S, \mathsf{spost}(C, P))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{svc}(\mathbf{skip}, P) &= \emptyset \\
\mathsf{svc}(x := e, P) &= \emptyset \\
\mathsf{svc}(\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f, P) &= \mathsf{svc}(S_t, P \wedge b) \cup \mathsf{svc}(S_f, P \wedge \neg b) \\
\mathsf{svc}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S, P) &= \{P \to I, \mathsf{spost}(S, I \wedge b) \to I\} \cup \mathsf{svc}(S, I \wedge b) \\
\mathsf{svc}(\mathbf{call}\ \mathbf{p}, P) &= \emptyset \\
\mathsf{svc}(C; S, P) &= \mathsf{svc}(C, P) \cup \mathsf{svc}(S, \mathsf{spost}(C, P))
\end{aligned}
$$

where $\overline{y}$ is a sequence of the auxiliary variables of $\mathbf{p}$
$\overline{x}$ is a sequence of the program variables occurring in $\mathbf{body}(\mathbf{p})$
$\overline{x_f}$ and $\overline{y_f}$ are sequences of fresh variables
The expression $t[\overline{e}/\overline{x}]$, with $\overline{x} = x_1, \ldots, x_n$ and $\overline{e} = e_1, \ldots, e_n$,
denotes the parallel substitution $t[e_1/x_1, \ldots, e_n/x_n]$

Figure 2: Verification conditions for blocks of commands with procedure calls

first and second components of the specification, even when the triple $\{P\}\,\underline{S}\,\{Q\}$ is not valid.

The well-known inference system of *Hoare logic* is *sound* with respect to the standard operational semantics of *While* programs in the sense that if the triple $\{P\}\,\underline{S}\,\{Q\}$ is derivable, then $\underline{S}$ is correct with respect to P and Q. Let $\vdash$ denote derivability using the axioms and rules of Hoare logic; then

$$\vdash \{P\}\,\underline{S}\,\{Q\} \quad \text{implies} \quad [\![\{P\}\,\underline{S}\,\{Q\}]\!] = \mathit{true}$$

In order to prove correctness axiomatically, one must use some strategy for constructing derivations in a backward manner (derivations are not unique for a given triple). Recall the *While* and *Consequence* rules of Hoare logic:

$$\frac{\{I \wedge b\}\,S\,\{I\}}{\{I\}\,\mathbf{while}\ b\ \mathbf{do}\ S\,\{I \wedge \neg b\}} \qquad \frac{\{P'\}\,S\,\{Q'\}}{\{P\}\,S\,\{Q\}} \quad \text{if } \models P \to P' \text{ and } \models Q' \to Q$$

To derive $\{P\}\,\mathbf{while}\ b\ \mathbf{do}\ S\,\{Q\}$ one has first to be able to find an appropriate invariant I of the loop in order to apply the *While* rule, and the first-order side conditions $P \to I$ and $I \wedge \neg b \to Q$ then have to be checked.

In practice, working program verification systems require users to provide the invariants as annotations in the code, and then employ a *backward propagation strategy* to construct derivations. In fact, these derivations do not even need to be explicitly constructed: an algorithm can be used that takes as input a piece of

annotated code together with a specification, and produces a set of first-order proof obligations ensuring that a derivation exists. Such an algorithm is usually known as a *verification conditions generator* (VCGen). A set of verification conditions for the annotated programs of Figure 1 is given as follows, where the functions wprec and wvc are defined in Figure 2:

$$\mathsf{VCG}(P,\ S,\ Q)\ =\ \{P \to \mathsf{wprec}(S, Q)\}\ \cup\ \mathsf{wvc}(S, Q)$$

In brief, the function wprec calculates the weakest precondition of a block, except in the case of loops. In this case, the function simply returns the annotated invariant (hopefully an approximation of the weakest precondition), and the function wvc collects, for every loop contained in the block, additional conditions required to establish that the annotation is indeed an invariant, and that it is sufficiently strong for the loop to attain the desired postcondition.

We make the following more detailed remarks:

- All the conditions in the set $\mathsf{VCG}(P,\ S,\ Q)$ should be proved valid to ensure the correctness of the program, see Proposition 1 below.
- This set is constructed following a strategy that propagates the postcondition Q backwards; the function wvc collects the side conditions of an implicit Hoare logic tree, in which rightmost branches are constructed before leftmost branches, without using the precondition P.
- When considering a block of the form $C\,;\,S$, the corresponding Hoare logic rule dictates that two derivations should be recursively considered for C and S. The backwards strategy first considers the block S with the given postcondition Q, and then the command C with the postcondition $\mathsf{wprec}(S, Q)$. Thus the function wprec guides the strategy by selecting an intermediate assertion propagated backwards from Q.
- The definition of wprec shares much with Dijkstra's *weakest liberal precondition* predicate transformer; the difference is that whereas a true weakest precondition is given as a least-fixpoint solution to a recursive equation, wprec simply makes use of the annotated loop invariant. Note that it may well be the case that this annotation is not in fact an invariant; even if it is an invariant, it is possible that it is not sufficiently strong to allow Q as a postcondition of the loop; on the other hand, the annotation does not need to be the weakest of all sufficiently strong invariants, and often is not.
- An additional verification condition is added to the set collected by wvc, stating that the precondition P must be stronger than the assertion propagated backward from Q through the block S.

This VCGen is *sound*: it can be proved that a Hoare logic tree with conclusion $\{P\}\,\underline{S}\,\{Q\}$ can be constructed that has exactly the assertions in the set $\mathsf{VCG}(P,\ S,\ Q)$ as side conditions; if these conditions are all valid then the tree is indeed a proof tree, and the triple is derivable in Hoare logic. Let $\models \mathcal{A}$, with $\mathcal{A}$ a set of first-order formulas, denote the fact that $\models A_i$ for every $A_i \in \mathcal{A}$ (in this context set union will be denoted by commas).

Proposition 1 (Soundness of VCGen)**.** *Let P, Q be assertions and S a command block. If $\models \mathsf{VCG}(P,\ S,\ Q)$ then $\vdash \{P\}\,\underline{S}\,\{Q\}$.*

Observe that the reverse implication does not hold in general: it may well be the case that a triple is derivable in Hoare logic, but the invariants that have been annotated result in a set of verification conditions that are not all valid (see previous remark on annotated invariants). This closely corresponds to what happens in practice when using a verification system: failure of the process may be caused by either errors in the initial code, or errors in the user-provided annotations. The reverse result does hold for programs that are correctly annotated in the following sense:

Definition 1. *A command block S is said to be* correctly annotated *with respect to a specification (P, Q) if whenever $\vdash \{P\}\,\underline{S}\,\{Q\}$, there exists a derivation of this triple that uses, in each application of the* While *rule, exactly the invariant that is annotated in the corresponding loop in S.*

Proposition 2 (Adequacy of VCGen)**.** *Let S be a correctly annotated command block with respect to (P, Q). If $\vdash \{P\}\,\underline{S}\,\{Q\}$ then $\models \mathsf{VCG}(P,\ S,\ Q)$.*

For more details on the generation of verification conditions for annotated programs and on how properties of the VCGens can be mechanically verified, see [16, 21]. A survey of related work can be found in [14].

2.2 Combining Forward and Backward Propagation

The set of VCs given above is not unique: there are equivalent sets of assertions that can be generated from the program and its intended specification. An alternative method is based on propagating the given precondition forward, using a strategy that constructs leftmost branches of the implicit proof trees before proceeding to the rightmost branches. The function that guides such a strategy is reminiscent of the *strongest postcondition* predicate transformer, instead of the previous weakest liberal precondition.

Rather than looking in detail at this alternative strategy, we will now consider a more general method, which allows the correctness of a given block of code to be verified by combining forward propagation of the precondition and backward propagation of the postcondition. Although there is nothing very sophisticated about this method, it has not as far as we know been spelled out before. Let $S = C_1 ; \ldots ; C_n$. We introduce the following notation, for $k \in \{0, \ldots, n\}$ (the functions spost and svc are defined in Figure 2):

$$\mathsf{wprec}^k(S, Q) = \mathsf{wprec}(C_k ; \ldots ; C_n, Q) \qquad \mathsf{wvc}^k(S, Q) = \mathsf{wvc}(C_k ; \ldots ; C_n, Q)$$
$$\mathsf{wprec}^{n+1}(S, Q) = Q \qquad \mathsf{wvc}^{n+1}(S, Q) = \emptyset$$

$$\mathsf{spost}^0(S, P) = P \qquad \mathsf{svc}^0(S, P) = \emptyset$$
$$\mathsf{spost}^k(S, P) = \mathsf{spost}(C_1 ; \ldots ; C_k, P) \qquad \mathsf{svc}^k(S, P) = \mathsf{svc}(C_1 ; \ldots ; C_k, P)$$

Then we define:

Definition 2 (Forward-backward Verification Conditions)**.**

$$\mathsf{VCG}^k(P,\, S,\, Q) \;=\; \mathsf{svc}^k(S, P) \;\cup\; \{\mathsf{spost}^k(S, P) \rightarrow \mathsf{wprec}^{k+1}(S, Q)\} \;\cup\; \mathsf{wvc}^{k+1}(S, Q)$$

Forward-backward VCs are obtained by forward propagation up to the k^{th} command and by backward propagation for subsequent commands, and this applies also to subblocks of these commands: forward propagation is used for subblocks of commands located before k, and backward propagation for those after k. The exclusively backward propagation VCGen is a special case: $\mathsf{VCG}(P,\, S,\, Q) = \mathsf{VCG}^0(P,\, S,\, Q)$. We will also use the notation $\overline{\mathsf{VCG}}(P,\, S,\, Q) = \mathsf{VCG}^n(P,\, S,\, Q)$, with n the length of S, for the forward propagation VCGen. The following lemma states some basic properties of VCs generated by forward, backward, and forward-backward propagation.

Lemma 1.

1. *If S is* $\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f$*, then*
   $\mathsf{VCG}(P,\, S,\, Q) = \mathsf{VCG}(P \wedge b,\, S_t,\, Q) \cup \mathsf{VCG}(P \wedge \neg b,\, S_f,\, Q)$ *and*
   $\overline{\mathsf{VCG}}(P,\, S,\, Q) = \overline{\mathsf{VCG}}(P \wedge b,\, S_t,\, Q) \cup \overline{\mathsf{VCG}}(P \wedge \neg b,\, S_f,\, Q)$
2. *If S is* $\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S_w$*, then*
   $\mathsf{VCG}(P,\, S,\, Q) = \{P \rightarrow I, I \wedge \neg b \rightarrow Q\} \cup \mathsf{VCG}(I \wedge b,\, S_w,\, I)$ *and*
   $\overline{\mathsf{VCG}}(P,\, S,\, Q) = \{P \rightarrow I, I \wedge \neg b \rightarrow Q\} \cup \overline{\mathsf{VCG}}(I \wedge b,\, S_w,\, I)$
3. *If S is $C_1 ; \ldots ; C_n$ with $n > 1$, then for $k \in \{0, \ldots, n\}$,*
   $\mathsf{VCG}^k(P,\, S,\, Q) = \overline{\mathsf{VCG}}(P,\, C_1 ; \ldots ; C_k,\, \mathsf{wprec}^{k+1}(S, Q))$
   $\qquad\qquad \cup\, \mathsf{VCG}(\mathsf{spost}^k(S, P),\, C_{k+1} ; \ldots ; C_n,\, Q)$
4. *If S consists of a single command C, then*
   $\models \mathsf{VCG}(P,\, S,\, Q)$ *iff* $\models \overline{\mathsf{VCG}}(P,\, S,\, Q)$

Proof. All results follow directly from the definitions, with the exception of (4) which is proved by structural induction, using (1) and (2) for the inductive cases. For the base cases we make use of the first-order logical equivalences $\forall x.\, \phi \rightarrow \psi \equiv \phi \rightarrow \forall x.\, \psi$ (if x does not occur free in ϕ) and $\forall x.\, \phi \rightarrow \psi \equiv (\exists x.\, \phi) \rightarrow \psi$ (if x does not occur free in ψ). □

As expected, one can equivalently use any value of k to generate verification conditions.

Proposition 3. *Let P, Q be assertions, $S = C_1 ; \ldots ; C_n$ a block of commands, and $k \in \{0, \ldots, n\}$. Then $\models \mathsf{VCG}(P,\, S,\, Q)$ iff $\models \mathsf{VCG}^k(P,\, S,\, Q)$.*

```
pre true
post x ≥ 0
proc abs =
  if x < 0 then x := −x else skip ;
  if c > 0 then c := c − 1 else skip
```

Figure 3: Example procedure: **abs**

Proof. Using the definitions one has

$$\begin{aligned}\mathsf{VCG}^k(P,\,S,\,Q) &= \mathsf{svc}^k(S,P) \cup \{\mathsf{spost}^k(S,P) \to \mathsf{wprec}^{k+1}(S,Q)\} \cup \mathsf{wvc}^{k+1}(S,Q)\\ &= \mathsf{svc}^k(S,P) \cup \mathsf{VCG}(\mathsf{spost}^k(S,P),\, C_{k+1},\, \mathsf{wprec}^{k+2}(S,Q)) \cup \mathsf{wvc}^{k+2}(S,Q)\end{aligned}$$

$$\begin{aligned}\mathsf{VCG}^{k+1}(P,\,S,\,Q) &= \mathsf{svc}^{k+1}(S,P) \cup \{\mathsf{spost}^{k+1}(S,P) \to \mathsf{wprec}^{k+2}(S,Q)\} \cup \mathsf{wvc}^{k+2}(S,Q)\\ &= \mathsf{svc}^k(S,P) \cup \overline{\mathsf{VCG}}(\mathsf{spost}^k(S,P),\, C_{k+1},\, \mathsf{wprec}^{k+2}(S,Q)) \cup \mathsf{wvc}^{k+2}(S,Q)\end{aligned}$$

And thus $\models \mathsf{VCG}^k(P,\,S,\,Q)$ iff $\models \mathsf{VCG}^{k+1}(P,\,S,\,Q)$ by Lemma 1(4). It now suffices to iterate this step starting with $k = 0$. □

Definition 2 does not specify all the possible sets of verification conditions of S with respect to (P, Q). In the rest of the paper we will investigate the problem of defining sets of VCs that combine forward and backward propagation in non-standard ways, using different combinations for different subblocks – for instance a command C in S that is being processed by forward propagation may contain subblocks that may be processed by backward (or a combination of forward and backward) propagation.

2.3 Example

Consider the very simple procedure defined in Figure 3 (taken from [12]). It calculates the absolute value of a number and decreases the value of a counter, as long as this value was initially positive (note that the contract does not describe this second part of the functionality). The code is a sequence of two conditional commands, and it contains no loops, thus wvc and svc return empty sets. We have three different sets of verification conditions (each consisting of a single condition), as follows:

$$\begin{aligned}&\mathsf{VCG}^0(\mathbf{pre}(\mathbf{abs}),\, \mathbf{body}(\mathbf{abs}),\, \mathbf{post}(\mathbf{abs}))\\ =\;& \{\mathsf{true} \to (x < 0 \to (c > 0 \to -x \geq 0) \wedge (\neg\, c > 0 \to -x \geq 0) \wedge\\ &\qquad (\neg\, x < 0 \to (c > 0 \to x \geq 0) \wedge (\neg\, c > 0 \to x \geq 0)))\}\\ &\mathsf{VCG}^1(\mathbf{pre}(\mathbf{abs}),\, \mathbf{body}(\mathbf{abs}),\, \mathbf{post}(\mathbf{abs}))\\ =\;& \{(\exists\, v.\, v < 0 \wedge x = -v) \vee (\neg\, x < 0) \to (c > 0 \to x \geq 0) \wedge (\neg\, c > 0 \to x \geq 0)\}\\ &\mathsf{VCG}^2(\mathbf{pre}(\mathbf{abs}),\, \mathbf{body}(\mathbf{abs}),\, \mathbf{post}(\mathbf{abs}))\\ =\;& \{(\exists\, w.\, ((\exists\, v.\, v < 0 \wedge x = -v) \vee (\neg\, x < 0)) \wedge w > 0 \wedge c = w - 1) \vee\\ &\quad (((\exists\, v.\, v < 0 \wedge x = -v) \vee (\neg\, x < 0)) \wedge \neg\, c > 0) \to x \geq 0\}\end{aligned}$$

2.4 Procedure Calls and Adaptation

Occurrences of variables in the assertions $\mathbf{pre}(\mathbf{p})$ and $\mathbf{post}(\mathbf{p})$ refer to their values in the pre-state and post-state of execution of the procedure $\mathbf{p}$ respectively; the use of *auxiliary variables* that occur in contracts (and possibly also in annotations, but not in the code) is essential, since they allow specifying how the output of a procedure is related to its input. For instance if $\mathbf{pre}(\mathbf{p})$ is $x = x_0 \wedge y = y_0$ and $\mathbf{post}(\mathbf{p})$ is $x = y_0 \wedge y = x_0$, this contract specifies that $\mathbf{p}$ swaps the values of the variables x and y.

The clauses in the definition of the functions wprec and spost for the case of the procedure call command (Figure 2) deserve the following two remarks:

- Similarly to the case of loops, they do not calculate the weakest precondition / strongest postcondition, since it may be the case that the code $\mathbf{body}(\mathbf{p})$ is not correct with respect to its contract, or that $\mathbf{pre}(\mathbf{p})$ is not the weakest precondition to ensure the postcondition $\mathbf{post}(\mathbf{p})$.

- The definition takes into account a phenomenon known as *adaptation*, which occurs in the presence of auxiliary variables. Although it is out of the scope of this paper to explain this in detail, it is easy to understand why the naive definition $\mathsf{wprec}(\mathbf{call}\,\mathbf{p}, Q) = \mathbf{pre}(\mathbf{p})$ would not work: it suffices to think of the above example where $\mathbf{pre}(\mathbf{p})$ is $x = x_0 \wedge y = y_0$. It would not make any sense to require the calling procedure to enforce $x = x_0$ when x_0 is an auxiliary variable, not even known in its scope. The reader is referred to [23] for details on adaptation.

2.5 Interprocedural Verification

A program Π is correct when the triple $\{\mathbf{pre}(\mathbf{p})\}\,\mathbf{call}\,\mathbf{p}\,\{\mathbf{post}(\mathbf{p})\}$ is valid for every $\mathbf{p} \in \mathcal{P}(\Pi)$. Our setting for the verification of programs consisting of several procedures builds on the principles prescribed by the software development methodology known as *design by contract* [26]. The idea is that each individual procedure can be verified by assuming that every procedure it invokes is correct with respect to its announced contract. If this is successfully done for every procedure in the program, then the program is correct. A set of verification conditions for Π is given as follows:

$$\mathsf{Verif}(\Pi) \;=\; \bigcup_{\mathbf{p}\in\mathcal{P}(\Pi)} \mathsf{VCG}(\mathbf{pre}(\mathbf{p}),\, \mathbf{body}(\mathbf{p}),\, \mathbf{post}(\mathbf{p}))$$

A set of VCs is generated for each procedure, assuming the correctness of all the procedures in the program, including itself. If all verification conditions are valid, correctness is established *simultaneously* for the entire set of procedures in the program, with all correctness assumptions dropped. Note that this only makes sense in the *partial correctness* setting in which we are working; the framework does not provide a mechanism for ensuring that the mutually recursive procedures in the set $\mathcal{P}(\Pi)$ terminate.

This simple framework formalizes the basis of many modern program verification tools [11, 2], which are mostly based on design by contract.

3 Labeled Control Flow Graphs

In this section we formalize a notion of control flow graph of a command block, and define two labelings of its edges, corresponding respectively to forward propagation and backward propagation of an assertion.

Definition 3 (Control Flow Graph)**.** *Given a block $S = C_1\,;\,\ldots\,;\,C_n$ of commands, we define simultaneously the* control flow graph *$CFG(S)$ of S (which is a directed acyclic graph), and the functions $IN(\cdot)$ and $OUT(\cdot)$ that associate to each command C_i with $i \in \{1,\ldots,n\}$, respectively its* input node *and its* ouput node *in $CFG(S)$, as follows:*

1. *The set of nodes of $CFG(S)$ contains two nodes with labels $START$ and END, and for each command C_i in S:*
   - *A single node with label C_i, if C_i is* **skip** *or an assignment command or a procedure call command. We set both $IN(C_i)$ and $OUT(C_i)$ to be this node.*
   - *Two nodes with labels* **if**(b) / **fi** *(resp.* **do**$(b)\{I\}$ / **od**$\{I\}$*), if C_i is* **if** b **then** S_t **else** S_f *(resp.* **while** b **do** $\{I\}$ S*). We set $IN(C_i)$ and $OUT(C_i)$ to be the former and the latter node respectively.*
2. *The set of edges of $CFG(S)$ contains an edge $(OUT(C_i), IN(C_{i+1}))$ for $i \in \{1,\ldots,n-1\}$, and two additional edges $(START, IN(C_1))$ and $(OUT(C_n), END)$.*
3. *Additionally for each command C_i in S that contains subblocks, we construct the corresponding* CFG *and graft it into $CFG(S)$ as follows:*
   - *If $C_i =$* **if** b **then** S_t **else** S_f*, we recursively construct $CFG(S_t)$ and $CFG(S_f)$, then remove their $START$ nodes and set the source of the dangling edges to be in both cases the node $IN(C_i)$, and similarly remove their END nodes and set the destination of the dangling edges to be the node $OUT(C_i)$.*
   - *If $C_i =$* **while** b **do** $\{I\}$ S*, we recursively construct $CFG(S)$, then remove its $START$ node and set the source of the dangling edge to be the node $IN(C_i)$, and similarly remove its END node and set the destination of the dangling edge to be the node $OUT(C_i)$.*

Clearly, to every subblock $\hat{S}$ of S corresponds a subgraph of $CFG(S)$ delimited by a pair of nodes with labels $START$ / END, or **if**(b) / **fi**, or **do**$(b)\{I\}$ / **od**$\{I\}$. An *execution path* of $CFG(S)$ is any path beginning in the node $START$ and ending in the node END. Each concrete execution of a program corresponds to one particular execution path in its CFG.

Definition 4 (Backward and forward propagation labelings)**.** *Given a block of commands S and an assertion Q, the functions $wplb^Q(\cdot)$ and $splb^Q(\cdot)$ assign a label to each edge of the graph $CFG(S)$.*

$wplb^Q(\cdot)$ is the backward propagation labeling. The label of the incoming edge e into the node END is $wplb^Q(e) = Q$, and for every node N in $CFG(S)$:

- *If N has an atomic command label C, with incoming edge i, and outgoing edge o, then we set $wplb^Q(i)$ to be $\mathsf{wprec}(C, wplb^Q(o))$.*
- *If the label of N is* **fi***, let i_t, i_f be the two incoming edges into N and o be the outgoing edge from N; then we set both $wplb^Q(i_t)$ and $wplb^Q(i_f)$ to be $wplb^Q(o)$.*
- *If the label of N is $\mathbf{if}(b)$, let i be the incoming edge into N and o_t, o_f be the two outgoing edges from N; then we set $wplb^Q(i)$ to be*
  $(b \to wplb^Q(o_t)) \wedge (\neg b \to wplb^Q(o_f))$.
- *If the label of N is $\mathbf{od}\{I\}$ and i is its incoming edge, then we set $wplb^Q(i)$ to be I.*
- *If the label of N is $\mathbf{do}(b)\{I\}$, let i be the incoming edge into N; then we set $wplb^Q(i)$ to be I.*

As for the forward propagation labeling $splb^Q(\cdot)$, the label of the outgoing edge e from the node $START$ is $splb^P(e) = P$, and for every node N in $CFG(S)$:

- *If N has an atomic command label C, with incoming edge i, and outgoing edge o, then we set $splb^P(o)$ to be $\mathsf{spost}(C, splb^P(i))$.*
- *If the label of N is $\mathbf{if}(b)$, let i be the incoming edge into N and o_t, o_f be the two outgoing edges from N; then we set $splb^P(o_t)$ to be $splb^P(i) \wedge b$, and $splb^P(o_f)$ to be $splb^P(i) \wedge \neg b$.*
- *If the label of N is* **fi***, let i_t, i_f be the two incoming edges into N and o be the outgoing edge from N; then we set $splb^P(o)$ to be $splb^P(i_t) \vee splb^P(i_f)$.*
- *If the label of N is $\mathbf{do}(b)\{I\}$, let o be the outgoing edge from N; then we set $splb^P(o)$ to be $I \wedge b$.*
- *If the label of N is $\mathbf{od}\{I\}$ and o its outgoing edge, then we set $splb^P(o)$ to be $I \wedge \neg b$.*

The following lemma states that these labelings do in fact propagate assertions as prescribed by wprec and spost.

Lemma 2. *Let $S = C_1 ; \ldots ; C_n$ be a block of commands and (P, Q) a specification. Then for every $k \in \{1, \ldots, n\}$,*

1. *$wplb^Q(i) = \mathsf{wprec}(C_k, wplb^Q(o))$ and $splb^P(o) = \mathsf{spost}(C_k, splb^P(i))$;*
2. *$wplb^Q(i) = \mathsf{wprec}^k(S, Q)$ and $splb^P(o) = \mathsf{spost}^k(S, P)$;*

where i, o, are respectively the edges of the graph $CFG(S)$ incoming into the node $IN(C_k)$ and outgoing from the node $OUT(C_k)$.

Proof. (1) and (2) are proved by mutual induction. The proofs are symmetric for both labelings, we focus on the statements involving $splb(\cdot)$.

(1) is established by structural induction on C_k using 2. and Definition 4.

(2) is proved by induction on k. For $k = 1$, $splb^P(START, IN(C_1)) = P$ by definition, and application of (1) yields $splb^P(OUT(C_1), IN(C_2)) = \mathsf{spost}(C_1, P)$; the inductive case also follows from (1).

□

4 Verification Graphs

Let us now see how sets of verification conditions can be generated from the above labelings.

Definition 5. *Let S be a program block, E the set of edges of the graph $CFG(S)$, and (P, Q) a specification; we define the* edge condition *of the edge $e \in E$ with respect to (P, Q) as $ec^{P,Q}(e) = splb^P(e) \to wplb^Q(e)$, and the set $\mathsf{EC}(P, S, Q)$ of* edge conditions *of S as*

$$\mathsf{EC}(P, S, Q) = \bigcup_{e \in E} ec^{P,Q}(e)$$

Let also $E_v \subseteq E$ and $\mathsf{EC}(E_v)$ be the corresponding set of conditions. We say that E_v is adequate for verification *whenever $\models \mathsf{EC}(E_v)$ iff $\models \mathsf{EC}(P, S, Q)$.*

The interest of the labeled graphs resides in the close relationship between edge conditions and verification conditions. The set of edge conditions contains *all* the equivalent sets of VCs of Definition 2, but also non-standard verification conditions, following our remark at the end of Section 2.2.

Proposition 4. *Let P, Q be assertions, $S = C_1 ; \ldots ; C_n$ a block of commands, and $k \in \{0, \ldots, n\}$; then*

$$\models \mathsf{VCG}^k(P,\, S,\, Q) \quad \textit{iff} \quad \models \mathsf{EC}\,(P, S, Q)$$

(See A for the proof of this proposition.)

If any standard set of VCs (as given by the definition) is valid, then so are all the edge conditions. It is less clear how to identify smaller, ideally minimal, sets of edge conditions that are adequate for verification, i.e. whose validity implies the validity of all other edge conditions, and thus of a standard set of VCs. Each such subset of $\mathsf{EC}\,(P, S, Q)$ corresponds to one particular verification strategy, combining forward and backward propagation. Our goal will now be to identify in precise terms these minimal sets of edge conditions that are adequate for verification. We do this by studying how the conditions of adjacent edges are related.

Lemma 3. *In the conditions of Lemma 2, with i the incoming edge into the node with label $IN(C_k)$, and o the outgoing edge from the node $OUT(C_k)$:*

1. *If C_k is not a loop command and does not contain loops as subcommands, then $\models ec^{P,Q}(i)$ iff $\models ec^{P,Q}(o)$*
2. *If C_k is* **if** b **then** S_t **else** S_f*, let o_t, o_f be the outgoing edges from $IN(C_k)$ and i_t, i_f the incoming edges into $OUT(C_k)$, corresponding to the* then *and* else *branch respectively. Then*
   $\models ec^{P,Q}(i)$ *iff* $\models ec^{P,Q}(o_t)$ *and* $\models ec^{P,Q}(o_f)$ *and*
   $\models ec^{P,Q}(o)$ *iff* $\models ec^{P,Q}(i_t)$ *and* $\models ec^{P,Q}(i_f)$

Proof. Applying Definition 4 and Lemma 2:

1. We need to prove the equivalence $splb^P(i) \rightarrow \mathsf{wprec}(C_k, wplb^Q(o)) \equiv \mathsf{spost}(C_k, splb^P(i)) \rightarrow wplb^Q(o)$. Notice that since C_k does not contain loops,
   $\mathsf{VCG}(splb^P(i),\, C_k,\, wplb^Q(o)) = \{splb^P(i) \rightarrow \mathsf{wprec}(C_k, wplb^Q(o))\}$ and $\overline{\mathsf{VCG}}(splb^P(i),\, C_k,\, wplb^Q(o)) = \{\mathsf{spost}(C_k, splb^P(i)) \rightarrow wplb^Q(o)\}$, and the equivalence follows from Lemma 1(4).
2. This case amounts to proving the following propositional equivalence:
   $splb^P(i) \rightarrow ((b \rightarrow wplb^Q(o_t)) \wedge (\neg b \rightarrow wplb^Q(o_f))) \quad \equiv$
   $(splb^P(i) \wedge b \rightarrow wplb^Q(o_t)) \quad \wedge \quad (splb^P(i) \wedge \neg b \rightarrow wplb^Q(o_f))$, and
   $(splb^P(i_t) \vee splb^P(i_f)) \rightarrow wplb^Q(o) \quad \equiv$
   $(splb^P(i_t) \rightarrow wplb^Q(o)) \quad \wedge \quad (splb^P(i_f) \rightarrow wplb^Q(o))$

□

Blocks without Loops Note that in the first point above, i and o are adjacent when C_k is **skip**, $x := e$, or **call q**. Thus in the case of atomic commands adjacent edges have equivalent conditions. For conditional commands not containing loops the (non-adjacent) incoming and outgoing edges of the subgraph have equivalent conditions, and moreover the conjunction of the conditions of the branching edges is equivalent to the adjacent edge's condition.

Proposition 5. *Let S be a block not containing loops and E_v a subset of edges of $CFG(S)$. Then E_v is adequate for verification iff for every execution path ϕ of $CFG(S)$, E_v contains an edge of ϕ.*

Proof. Since S contains no loops we have that $\mathsf{VCG}(P,\, S,\, Q) = \{P \rightarrow \mathsf{wprec}(S, Q)\}$. Following Proposition 4, E_v is adequate for verification whenever $\models \mathsf{EC}\,(E_v)$ iff $\models P \rightarrow \mathsf{wprec}(S, Q)$. Now note that the latter assertion is the condition of the edge with origin in the node $START$. But Lemma 3 implies that if S does not contain loops and conditionals (there is a single path) all edge conditions are equivalent, and if it does contain conditionals, the edge condition in the path before the conditional is equivalent to the condition in the path after the conditional, and to the conjunction of two edge conditions, one for each branch path. Thus whatever the branching structure of S may be, the conjunction of the conditions in the set E_v is equivalent to $P \rightarrow \mathsf{wprec}(S, Q)$ iff E_v contains, for every execution path of the graph, at least one edge that is part of that path. □

The *minimal* sets of edges that are adequate for verification contain *exactly one* edge of every execution path.

Blocks with Loops Recall that each loop is represented by a pair of nodes labeled $\mathbf{do}(b)\{I\}$, $\mathbf{od}\{I\}$, and a set of paths from the former to the latter, corresponding to the loop's body. Consider the subgraph

$$A \ \rightarrow^{\cdot^{\cdot^{\cdot}}}{}^{\phi_1} \rightarrow \ \mathbf{do}(b)\{I\} \ \rightarrow^{\cdot^{\cdot^{\cdot}}}{}^{\phi_2} \rightarrow \ \mathbf{od}\{I\} \ \rightarrow^{\cdot^{\cdot^{\cdot}}}{}^{\phi_3} \rightarrow \ B$$

Then the edge condition of ϕ_1 ensures loop initialization, ϕ_2 ensures invariant preservation, and ϕ_3 ensures that the postcondition is granted by the invariant upon termination. Unlike the case of atomic or branching nodes, in the presence of $\mathbf{do}(b)\{I\}$, $\mathbf{od}\{I\}$ nodes it is necessary to establish independently the validity of these edge conditions, since a result similar to Lemma 3 does not hold. Thus in general a set of edges E_v is adequate for verification iff for every execution path ϕ of $CFG(S)$,

- if ϕ does not traverse a loop command, E_v contains an edge of ϕ;
- for every pair of nodes labeled $\mathbf{do}(b)\{I\}$ / $\mathbf{od}\{I\}$ crossed by ϕ, E_v contains three edges of ϕ, one before the first node, a second edge between both nodes, and a third after the second node.

Procedures and Programs The above discussion can be carried over to the scope of procedures and programs as follows.

Definition 6 (Verification Graph)**.** *Let* $\mathbf{p} \in \mathcal{P}(\Pi)$ *with* Π *a program. Its* verification graph *is the graph* $CFG(\mathbf{body}(\mathbf{p}))$*, equipped with the labeling that assigns to each edge* e *the condition* $ec^{\mathbf{pre}(\mathbf{p}),\mathbf{post}(\mathbf{p})}(e)$*. We will denote by* $\mathsf{EC}(\mathbf{p})$ *the set of edge conditions* $\mathsf{EC}(\mathbf{pre}(\mathbf{p}), \mathbf{body}(\mathbf{p}), \mathbf{post}(\mathbf{p}))$*.*

Naturally, we may also speak of the set of edge conditions of a program Π, defined as $\mathsf{EC}(\Pi) = \bigcup_{\mathbf{p}\in\mathcal{P}(\Pi)} \mathsf{EC}(\mathbf{p})$. It is a straightforward consequence of Proposition 4 that $\models \mathsf{Verif}(\Pi)$ iff $\models \mathsf{EC}(\Pi)$.

Example Consider again the procedure of Figure 3. Its CFG is shown in Figure 4 (left), and the set of edge conditions $\mathsf{EC}(\mathbf{abs})$ consists of the following, where we have applied the simplifications $\mathsf{true} \wedge P \equiv P$ and $\mathsf{true} \rightarrow Q \equiv Q$:

$$\begin{aligned}
ec^{\mathsf{true},x\geq 0}(e_1) &= (x<0 \rightarrow (c>0 \rightarrow -x \geq 0) \wedge (\neg c>0 \rightarrow -x \geq 0)) \wedge \\
&\quad (\neg x<0 \rightarrow (c>0 \rightarrow x \geq 0) \wedge (\neg c>0 \rightarrow x \geq 0)) \\
ec^{\mathsf{true},x\geq 0}(e_2) &= x<0 \rightarrow (c>0 \rightarrow -x \geq 0) \wedge (\neg c>0 \rightarrow -x \geq 0) \\
ec^{\mathsf{true},x\geq 0}(e_3) &= (\exists v.\, v<0 \wedge x=-v) \rightarrow (c>0 \rightarrow x \geq 0) \wedge (\neg c>0 \rightarrow x \geq 0) \\
ec^{\mathsf{true},x\geq 0}(e_4) &= \neg x<0 \rightarrow (c>0 \rightarrow x \geq 0) \wedge (\neg c>0 \rightarrow x \geq 0) \\
ec^{\mathsf{true},x\geq 0}(e_5) &= (\exists v.\, v<0 \wedge x=-v) \vee (\neg x<0) \rightarrow (c>0 \rightarrow x \geq 0) \wedge (\neg c>0 \rightarrow x \geq 0) \\
ec^{\mathsf{true},x\geq 0}(e_6) &= ((\exists v.\, v<0 \wedge x=-v) \vee (\neg x<0)) \wedge c>0 \rightarrow x \geq 0 \\
ec^{\mathsf{true},x\geq 0}(e_7) &= \exists w.\, ((\exists v.\, v<0 \wedge x=-v) \vee (\neg x<0)) \wedge w>0 \wedge c=w-1 \rightarrow x \geq 0 \\
ec^{\mathsf{true},x\geq 0}(e_8) &= ((\exists v.\, v<0 \wedge x=-v) \vee (\neg x<0)) \wedge \neg c>0 \rightarrow x \geq 0 \\
ec^{\mathsf{true},x\geq 0}(e_9) &= (\exists w.\, ((\exists v.\, v<0 \wedge x=-v) \vee (\neg x<0)) \wedge w>0 \wedge c=w-1) \vee \\
&\quad (((\exists v.\, v<0 \wedge x=-v) \vee (\neg x<0)) \wedge \neg c>0) \rightarrow x \geq 0
\end{aligned}$$

Following Proposition 5, to verify the correctness of the procedure it suffices to check the validity of the edge conditions of one of the following:

- e_1, or
- e_2 or e_3, and e_4, or
- e_5, or
- e_6 or e_7, and e_8, or
- e_9

While extremely simple, this example illustrates an efficiency problem of verification conditions generated using the forward and backward propagation functions wprec and spost of Figure 2. The problem, first identified by Flanagan and Saxe [12] in the use of the weakest liberal precondition predicate transformer for the guarded commands language, has to do with branching and the fact that the postcondition is duplicated in

$$\mathsf{wprec}(\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f, Q) = (b \rightarrow \mathsf{wprec}(S_t, Q)) \wedge (\neg b \rightarrow \mathsf{wprec}(S_f, Q))$$

In $ec^{\mathsf{true},x\geq 0}(e_1)$ the postcondition $x \geq 0$ has been duplicated twice, and each copy occurs in a conjunct of the formula, which means that the theorem prover would in practice have to prove 4 conditions. In general, in a block S consisting of a sequence of n conditional commands, there would occur 2^n copies of

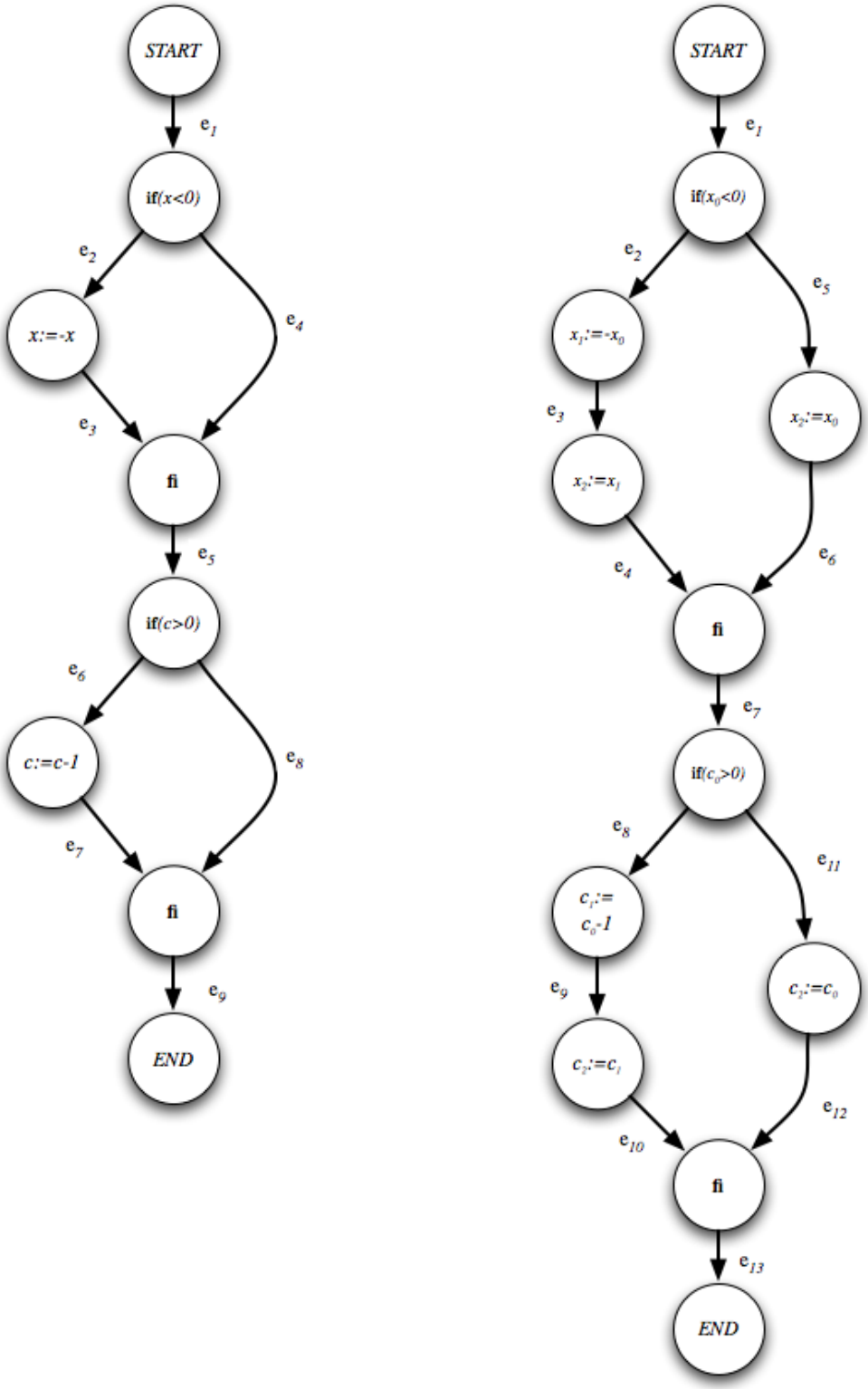


Figure 4: Control flow graphs of procedure **abs**

the postcondition Q in the assertion $\mathsf{wprec}(S, Q)$, an exponential number of conditions to be checked (on the length of the program).

The problem also occurs with forward propagation with spost – in our example the propagated precondition true does not occur 4 times in $ec^{\mathsf{true}, x \geq 0}(e_9)$ because we have simplified $\mathsf{true} \wedge P$ to P, but observe that the condition $x < 0$ of the first conditional, which is introduced in the assertion being propagated forward, already appears 4 times (although half of these disguised as v), and would in general appear an exponential number of times.

To sum up, forward propagation may create an exponential number of copies of the precondition, and backward propagation an exponential number of copies of the postcondition (and in both cases, also an exponential number of copies of the boolean conditions of the block). Although this problem cannot in general be solved by combining the use of wprec and spost, the example shows that different strategies may lead to very different verification conditions, from the point of view of the duplication of assertions.

Related Work In the context of the verification of reactive systems, Manna and colleagues have proposed the notion of *temporal verification diagram* [8] to represent a proof that a system enjoys a given property, expressed as a temporal logic formula. The idea is that such a diagram, whose edges are labelled by sets of transitions, corresponds to an approximation of the computations of a transition system. A set of (first order) verification conditions is produced from the diagram such that, if all VCs are valid, the system is guaranteed to satisfy the property under consideration.

5 Efficient Verification Conditions

The efficiency problem identified in the previous example has been addressed, and it is now well-known how the exponential growth of VCs can be eliminated. For backward propagation this was first studied in [12], and then simplified in [24]. More recently [18] it was shown that as a side effect, the solution allows for strongest postconditions to be computed dispensing with the introduction of existential quantifiers, and reveals a strong symmetry between forward and backward propagation calculations. In this section we show how assertions can be propagated forward and backward efficiently, by adapting to our language the solution proposed in the previous papers (originally developed in a guarded commands setting). We then go on to show how our results of the previous sections remain valid and useful in the new setting – all that it takes is a redefinition of the *wplb* and *splb* labelings.

Propagating assertions efficiently requires first converting the code to a so-called *passive form*, which involves creating multiple (indexed) versions of each variable.

Definition 7 (Passive Procedure)**.** *A procedure* $\mathbf{p}$ *is said to be* passive *if*

- *variables are assigned at most once in every execution path of* $\mathbf{body}(\mathbf{p})$*;*
- *the variables occurring in* $\mathbf{pre}(\mathbf{p})$ *are not assigned at all.*

Note that whenever $\mathsf{spost}(x := e, P)$ is calculated for generating the verification conditions of a passive form procedure, x will not occur in P. We will additionally assume that x does not occur in e in any instruction of the form $x := e$. Even though the above notion allows for this (as long as x has not been assigned before and does not occur in $\mathbf{pre}(\mathbf{p})$), such occurrences can easily be eliminated. Thus we have that

$$\begin{aligned} \mathsf{spost}(x := e, P) &\equiv P \wedge x = e \\ \mathsf{wprec}(x := e, Q) &\equiv x = e \rightarrow Q \end{aligned}$$

Note that the basic equivalence $\mathsf{spost}(C, P) \rightarrow Q \equiv P \rightarrow \mathsf{wprec}(C, Q)$, which was essential for proving Lemma 1, Proposition 3, and Lemma 3, becomes very obvious for passive assignments.

Example Let us return to our running example procedure **abs**. A passive form of the procedure is shown in Figure 5; its CFG is depicted in Figure 4 (right), and a selection of edge conditions is given in Figure 6 (top).

Observe that the effect of calculating VCs without the burden of existential quantifiers is quite dramatic and allows for a clearer appreciation of the differences between conditions obtained at different points of the CFG. But the efficiency problem has still not been attacked (in the example, there are still four copies of $x_2 \geq 0$ in the edge condition of e_1). The crucial observation is that whereas for general blocks of commands *assertions must be propagated through the structure of the commands*, for a passive block S it suffices to combine them with a formula that depends only on S.

pre true
post $x_2 \geq 0$
proc abs =
  if $x_0 < 0$ **then** $x_1 := -x_0$; $x_2 := x_1$
    else $x_2 := x_0$;
  if $c_0 > 0$ **then** $c_1 := c_0 - 1$; $c_2 := c_1$
    else $c_2 := c_0$

Figure 5: Example passive procedure: **abs**

$$
\begin{aligned}
ec^{\mathsf{true},x_2\geq 0}\,(e_1) &= (x_0 < 0 \to x_1 = -x_0 \to x_2 = x_1 \to \\
&\quad (c_0 > 0 \to c_1 = c_0 - 1 \to c_2 = c_1 \to x_2 \geq 0) \wedge (\neg\, c_0 > 0 \to c_2 = c_0 \to x_2 \geq 0)) \wedge \\
&\quad (\neg\, x_0 < 0 \to x_2 = x_0 \to \\
&\quad (c_0 > 0 \to c_1 = c_0 - 1 \to c_2 = c_1 \to x_2 \geq 0) \wedge (\neg\, c_0 > 0 \to c_2 = c_0 \to x_2 \geq 0)) \\
ec^{\mathsf{true},x_2\geq 0}\,(e_4) &= x_0 < 0 \wedge x_1 = -x_0 \wedge x_2 = x_1 \to \\
&\quad (c_0 > 0 \to c_1 = c_0 - 1 \to c_2 = c_1 \to x_2 \geq 0) \wedge (\neg\, c_0 > 0 \to c_2 = c_0 \to x_2 \geq 0) \\
ec^{\mathsf{true},x_2\geq 0}\,(e_6) &= \neg\, x_0 < 0 \wedge x_2 = x_0 \to \\
&\quad (c_0 > 0 \to c_1 = c_0 - 1 \to c_2 = c_1 \to x_2 \geq 0) \wedge (\neg\, c_0 > 0 \to c_2 = c_0 \to x_2 \geq 0) \\
ec^{\mathsf{true},x_2\geq 0}\,(e_7) &= (x_0 < 0 \wedge x_1 = -x_0 \wedge x_2 = x_1) \vee (\neg\, x_0 < 0 \wedge x_2 = x_0) \to \\
&\quad (c_0 > 0 \to c_1 = c_0 - 1 \to c_2 = c_1 \to x_2 \geq 0) \wedge (\neg\, c_0 > 0 \to c_2 = c_0 \to x_2 \geq 0) \\
ec^{\mathsf{true},x_2\geq 0}\,(e_{13}) &= (((x_0 < 0 \wedge x_1 = -x_0 \wedge x_2 = x_1) \vee (\neg\, x_0 < 0 \wedge x_2 = x_0)) \wedge c_0 > 0 \wedge c_1 = c_0 - 1 \wedge c_2 = c_1) \vee \\
&\quad (((x_0 < 0 \wedge x_1 = -x_0 \wedge x_2 = x_1) \vee (\neg\, x_0 < 0 \wedge x_2 = x_0)) \wedge \neg\, c_0 > 0 \wedge c_2 = c_0) \to \\
&\quad x_2 \geq 0
\end{aligned}
$$

$$
\begin{aligned}
ec^{\mathsf{true},x_2\geq 0}\,(e_1) &= (x_0 < 0 \wedge x_1 = -x_0 \wedge x_2 = x_1) \vee (\neg\, x_0 < 0 \wedge x_2 = x_0) \to \\
&\quad (c_0 > 0 \wedge c_1 = c_0 - 1 \wedge c_2 = c_1) \vee (\neg\, c_0 > 0 \wedge c_2 = c_0) \to x_2 \geq 0 \\
ec^{\mathsf{true},x_2\geq 0}\,(e_4) &= x_0 < 0 \wedge x_1 = -x_0 \wedge x_2 = x_1 \to \\
&\quad (c_0 > 0 \wedge c_1 = c_0 - 1 \wedge c_2 = c_1) \vee (\neg\, c_0 > 0 \wedge c_2 = c_0) \to x_2 \geq 0 \\
ec^{\mathsf{true},x_2\geq 0}\,(e_6) &= \neg\, x_0 < 0 \wedge x_2 = x_0 \to \\
&\quad (c_0 > 0 \wedge c_1 = c_0 - 1 \wedge c_2 = c_1) \vee (\neg\, c_0 > 0 \wedge c_2 = c_0) \to x_2 \geq 0 \\
ec^{\mathsf{true},x_2\geq 0}\,(e_7) &= (x_0 < 0 \wedge x_1 = -x_0 \wedge x_2 = x_1) \vee (\neg\, x_0 < 0 \wedge x_2 = x_0) \to \\
&\quad (c_0 > 0 \wedge c_1 = c_0 - 1 \wedge c_2 = c_1) \vee (\neg\, c_0 > 0 \wedge c_2 = c_0) \to x_2 \geq 0 \\
ec^{\mathsf{true},x_2\geq 0}\,(e_{13}) &= ((x_0 < 0 \wedge x_1 = -x_0 \wedge x_2 = x_1) \vee (\neg\, x_0 < 0 \wedge x_2 = x_0)) \wedge \\
&\quad ((c_0 > 0 \wedge c_1 = c_0 - 1 \wedge c_2 = c_1) \vee (\neg\, c_0 > 0 \wedge c_2 = c_0)) \;\to\; x_2 \geq 0
\end{aligned}
$$

Figure 6: Edge conditions for passive version of procedure **abs** (top), and the same conditions calculated efficiently using Lemma 4 (bottom).

$$
\begin{aligned}
\mathsf{spost}(\mathbf{skip}, P) &= P \\
\mathsf{spost}(x := e, P) &= P \wedge x = e \\
\mathsf{spost}(\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f, P) &= \mathsf{let}\ P \wedge b \wedge P_t \equiv \mathsf{spost}(S_t, P \wedge b) \\
&\qquad\quad P \wedge \neg b \wedge P_f \equiv \mathsf{spost}(S_f, P \wedge \neg b) \\
&\quad\ \mathsf{in}\ P \wedge ((b \wedge P_t) \vee (\neg b \wedge P_f)) \\
\mathsf{spost}(C\,;\, S, P) &= \mathsf{spost}(S, \mathsf{spost}(C, P)) \\
\\
\mathsf{wprec}(\mathbf{skip}, Q) &= Q \\
\mathsf{wprec}(x := e, Q) &= x = e \to Q \\
\mathsf{wprec}(\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f, Q) &= \mathsf{let}\ Q_t \to Q \equiv \mathsf{wprec}(S_t, Q) \\
&\qquad\quad Q_f \to Q \equiv \mathsf{wprec}(S_f, Q) \\
&\quad\ \mathsf{in}\ (b \wedge Q_t) \vee (\neg b \wedge Q_f) \to Q \\
\mathsf{wprec}(C\,;\, S, Q) &= \mathsf{wprec}(C, \mathsf{wprec}(S, Q))
\end{aligned}
$$

Figure 7: Forward and backward propagation of assertions for passive programs

Definition 8. *We define the formula* $\mathsf{F}(S)$ *of a command block* S *as follows:*

$$
\begin{aligned}
\mathsf{F}(\mathbf{skip}) &= \mathsf{true} \\
\mathsf{F}(x := e) &= x = e \\
\mathsf{F}(\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f) &= (b \wedge \mathsf{F}(S_t)) \vee (\neg b \wedge \mathsf{F}(S_f)) \\
\mathsf{F}(C\,;\, S) &= \mathsf{F}(C) \wedge \mathsf{F}(S)
\end{aligned}
$$

The formula $\mathsf{F}(S)$ is a logical characterization of the block S, of linear size on the size of S. The following lemma states that assertions can be propagated in either direction through a passive block without further traversals of the code, which means that the conditions $\mathsf{spost}(S, P)$ and $\mathsf{wprec}(S, Q)$ are also of linear size, containing a single copy of P or Q.

Lemma 4. *In the context of a passive procedure without iteration or procedure calls, forward and backward propagation can be calculated as follows:*

$$
\begin{aligned}
\mathsf{spost}(S, P) &\equiv P \wedge \mathsf{F}(S) \\
\mathsf{wprec}(S, Q) &\equiv \mathsf{F}(S) \to Q
\end{aligned}
$$

Proof. By induction on the structure of S (taking into account the above discussion for the base case $x := e$). □

Since Proposition 3 still holds, forward-backward VCs (following Definition 2) can be computed efficiently for passive programs. Observe however that it does not seem to be possible to calculate efficient VCs from verification graphs. Recall that for a conditional command, the graph should enjoy the fundamental property that the edge condition of the incoming edge is equivalent to the conjunction of the conditions of the two branches. Now, the efficient VCs are computed without propagating the assertion P inside the conditional command's subgraph, which clearly contradicts the former requirement since P would not be part of the branch edges' conditions.

But the problem has a solution: in fact it is harmless to duplicate assertions as long as the duplicated copies are reduced to a single copy once they have been propagated through both branches of the conditional. The alternative definitions of spost and wprec in Figure 7 incorporate this idea; using Lemma 4 it is straightforward to prove that for passive programs they are equivalent to the original definitions. It is now possible to modify our CFG labeling functions *wplb* and *splb* for these definitions, in a way that preserves the validity of Lemma 3 and Proposition 5.

Definition 9 (Efficient backward and forward propagation labelings)**.** *Given a block* S *and an assertion* Q*, the functions* $wplb_e^Q(\cdot)$ *and* $splb_e^Q(\cdot)$ *assign a label to each edge of the graph* $CFG(S)$ *as follows.*

The backward propagation label of the incoming edge e *into the node END is* $wplb_e^Q(e) = Q$*, and for every node* N *in* $CFG(S)$*:*

- *If* N *is* **skip***, with* i *(*o*) its incoming (outgoing) edge, we set* $wplb_e^Q(i)$ *to be* $wplb_e^Q(o)$*.*

– *If N has label $x := e$, with incoming edge i, and outgoing edge o, then we set $wplb_e^Q(i)$ to be $x = e \rightarrow wplb_e^Q(o)$.*
– *If the label of N is* **fi**, *let i_t, i_f be the two incoming edges into N and o be the outgoing edge from N; then we set both $wplb_e^Q(i_t)$ and $wplb_e^Q(i_f)$ to be $wplb_e^Q(o)$.*
– *If the label of N is* **if**(b), *let i be the incoming edge and o_t, o_f the two outgoing edges from N, with $wplb_e^Q(o_t) \equiv Q_t \rightarrow Q$ and $wplb_e^Q(o_f) \equiv Q_f \rightarrow Q$; then we set $wplb_e^Q(i)$ to be $(b \wedge Q_t) \vee (\neg b \wedge Q_f) \rightarrow Q$.*

Concerning forward propagation, the label of the outgoing edge e from the node $START$ is $splb_e^P(e) = P$, and for every node N in $CFG(S)$:

– *If N is* **skip**, *with i (o) its incoming (outgoing) edge, we set $splb_e^P(o)$ to be $splb_e^P(i)$.*
– *If N has label $x := e$, with incoming edge i, and outgoing edge o, then we set $splb_e^P(o)$ to be $splb_e^P(i) \wedge x = e$.*
– *If the label of N is* **if**(b), *let i be the incoming edge into N and o_t, o_f be the two outgoing edges from N; then we set $splb_e^P(o_t)$ to be $splb_e^P(i) \wedge b$, and $splb_e^P(o_f)$ to be $splb_e^P(i) \wedge \neg b$.*
– *If the label of N is* **fi**, *let i_t, i_f be the two incoming edges into N, with $splb_e^P(i_t) \equiv P \wedge b \wedge P_t$ and $splb_e^P(i_f) \equiv P \wedge \neg b \wedge P_f$, and o be the outgoing edge from N. Then we set $splb_e^P(o)$ to be $P \wedge ((b \wedge P_t) \vee (\neg b \wedge P_f))$.*

Turning back to our example, a selection of edge conditions calculated with the efficient propagation labelings is also shown in Figure 6 (bottom). It is immediate to see that the different sets of verification conditions obtained become very similar for passive programs: comparing the conditions of e_1, e_7, and e_{13} in the example clearly shows this. Also, e_4 and e_6 together are equivalent to the previous assertions, confirming that the graph can still be used to split conditions (see below).

We end the section with a note on the conversion of programs to passive form. This conversion naturally generates a great number of different variables. In [12], the authors state that the conversion increases code size by 30%, but the benefits in terms of the size of VCs and proof time can be enormous for "complex" code (VC size 0.1–10% of original, proof time 2–50% of the original). A more recent paper [18] proposes an improved conversion algorithm, and shows that "version-optimal" passive forms (that use the minimal possible number of variables) can be computed in linear time; the authors also consider other notions of optimality as well as algorithms for obtaining passive forms.

The conversion of procedure calls and loops to passive form (and VC generation for them) is not so well documented in the literature but is used in practice for instance in the Boogie tool [2]. In passive form the treatment of procedure calls becomes in fact easier since the use of auxiliary variables seems to be unnecessary. In the guarded commands setting, programs containing loops with invariants are typically converted to loopless programs that, while not equivalent to the original, are such that their correctness implies the correctness of the original, see for instance [3].

6 Applications of Verification Graphs

Propagating assertions along the graph is not computationally costly compared with the cost of automatic proof, and although verification graphs contain redundancy (since many equivalent edge conditions may be present), this is not reflected in the proof process itself, since a precise criterion can be used to select sets of conditions not containing redundancy.

Even so, it is possible to optimize the construction of verification graphs to reduce redundancy: for a given verification strategy based on graphs, assertions can be propagated lazily. For instance, with a forward propagation strategy, the **splb** labeling needs not be propagated at all since only the first edge condition in each path is required. With a user-directed strategy, in which the user manually selects the edges in the graph whose conditions will be checked (for instance with the help of a visual front-end), the propagation can be directed by the user's selection on demand. The result is that only one of the labelings is calculated for each edge, with the exception of edges whose conditions will be checked.

In this section we consider some applications of verification graphs. The first application is a strategy that *does* introduce redundancy in the proof process, with the goal of identifying error traces when the program is not correct. Other applications considered include VC splitting (to improve automatic proofs) and interactive verification.

6.1 Automatic Error Path Discovery

An alternative approach to generating verification conditions is based on *symbolic execution* [28]. For passive programs symbolic execution is very closely related to strongest postcondition calculations. But a defining characteristic of symbolic execution is that it generates one formula *for each execution path* of the program.

Symbolic execution VCs with respect to a specification (P, Q) can be generated in a straightforward manner. For passive programs without loops it suffices to traverse the CFG from $START$ to END, constructing a conjunctive formula as follows: let P initially be the given precondition. For each node with label $x := e$ crossed, let P become $P \wedge x = e$; for each node with label **if**(b) crossed towards the *then* (resp. *else*) branch, let P become $P \wedge b$ (resp. $P \wedge \neg b$). When END is reached, generate the verification condition $P \rightarrow Q$. Repeat until all paths have been traversed (using a depth-first strategy).

For the example of Figure 4 (right), this would result in the following set:

$$\begin{array}{ll}
\text{Path 1:} & x_0 < 0 \wedge x_1 = -x_0 \wedge x_2 = x_1 \wedge c_0 > 0 \wedge c_1 = c_0 - 1 \wedge c_2 = c_1 \rightarrow x_2 \geq 0 \\
\text{Path 2:} & x_0 < 0 \wedge x_1 = -x_0 \wedge x_2 = x_1 \wedge \neg c_0 > 0 \wedge c_2 = c_0 \rightarrow x_2 \geq 0 \\
\text{Path 3:} & \neg x_0 < 0 \wedge x_2 = x_0 \wedge c_0 > 0 \wedge c_1 = c_0 - 1 \wedge c_2 = c_1 \rightarrow x_2 \geq 0 \\
\text{Path 4:} & \neg x_0 < 0 \wedge x_2 = x_0 \wedge \neg c_0 > 0 \wedge c_2 = c_0 \rightarrow x_2 \geq 0
\end{array}$$

Of course, this method may generate an exponential number of VCs (and copies of the postcondition Q) on the length of the block. But it does have one significant positive aspect, which is the fact that there is a direct association between VCs and error traces: if one of the above conditions were shown to be invalid, it would immediately be known which execution path of the program produced an error; moreover, the number of erroneous paths would also be known. This is not so in our present framework of standard verification condition generators: in the example, the formulas $ec^{\mathsf{true}, x_2 \geq 0}(e_1)$, $ec^{\mathsf{true}, x_2 \geq 0}(e_7)$ or $ec^{\mathsf{true}, x_2 \geq 0}(e_{13})$, while much more efficient to check than the above set, are not associated with one concrete execution path.

State of the art VCGens solve this problem by instrumenting the VCs with additional labels and reading back the counter-examples generated by the automatic provers, which can then be mapped to execution traces. Verification graphs offer an alternative solution to this problem, which can be used with any prover, even when counter-examples are not available. Recall that the validity of the condition of an edge e implies that there is no error in any of the execution paths containing e; if the condition is invalid this means that *at least one* of the execution paths containing e is erroneous. Thus it is straightforward to conceive an algorithm for identifying error traces.

The algorithm identifies error paths by performing a depth-first traversal of the verification graph and invoking an external proof tool. Whenever a branching node with label **if**(b) is met, the algorithm first follows the *then* branch, and will later follow the *else* branch. For each branch explored, the condition of the first edge is checked. If it is valid, the traversal backtracks, since no execution path containing the edge produces errors. If not, the traversal is continued in order to identify the error paths containing the present edge. An error path is found when END is reached (or when no more **if** nodes can be met from the current node to END).

In our example, if the edge condition of edge 1 (the most efficient that can be generated) is valid, then we are done. Supposing it were not valid, we would try to find the error path(s) by checking the condition of edge 2. If this were valid, both paths corresponding to $x_0 < 0$ (paths 1 and 2 above) would be error-free, and e_5 would certainly not be valid since the error would have to be in one of the paths corresponding to $x_0 \geq 0$. Checking the condition of e_8 would then identify path 3 or 4 (or both) as violating the postcondition.

Note that even though the number of execution paths is exponential, this technique is feasible if the number of error paths is small, since each condition checked as valid will reduce by half the number of paths left to explore. In particular, if a single error path exists, finding it requires checking a linear number of conditions on the length of the block. Naturally, the program should first be transformed into passive form, so that the technique can be used with the optimized edge conditions of Section 5.

6.2 Verification Condition Splitting

For the case of verification conditions based on weakest preconditions, it has been shown [25] that splitting VCs (i.e. having a bigger number of smaller conditions) can in certain circumstances lead to substantial improvements with respect to the performance of an SMT solver, and also with respect to the quality of the error messages produced when verification fails. The authors reach the conclusion that some splitting may dramatically improve the running time of the prover (bringing it to reasonable values when it was initially unfeasible), but too much splitting may have a negative effect.

While we have not evaluated experimentally the advantages and adequate degrees of splitting, verification graphs and Proposition 5 offer a method for identifying split versions of a procedure's VCs, when forward propagation and backward propagation are combined. The method is of course compatible with the optimizations allowed by passive programs.

In particular the *dynamic splitting* strategy suggested in [25] prescribes that one should first try to prove a single VC for the entire block. The prover is given a time-out limit, after which the VC should be split

and the two resulting VCs sent to the prover (again with a time-out limit), and so on. This can be readily formulated as a forward or a backward splitting strategy on the verification graph: one starts with the edge condition of the edge with origin $START$, or of the edge with destination END, and follows the branching structure of the CFG in the forward or backward direction.

But the graph would also allow one to identify the points of the program where the level of splitting is maximal. In the example of Figure 4, it is indifferent to split the VCs in the first or in the second conditional, but if one of the conditionals contained additional branching, it should take priority for maximal splitting, and this could be identified by a simple graph algorithm.

6.3 Interactive Verification

The labeled CFG may be used as the basis for a visual interactive verification tool that displays the graph and assists the user in different tasks:

User-guided Verification Condition Generation The user may be given the possibility to select edge conditions to be sent to the prover, and the tool can help keep track of valid and invalid conditions. Human understanding allows for an easier identification of error paths, and we speculate that VC splitting, with the goal of improving the performance of the prover, may also profit from human intervention.

We have developed a prototype tool thats supports visualization, interactive selection of edge conditions, and coloring of edges to allow for the identification of error paths. The tool is described in B.

Online Annotation Editing It makes sense for an interactive tool to allow annotations to be modified online. But since in our framework contracts and loop invariants are part of the programs, modifying annotations corresponds in fact to changing the program under consideration. Observe however that *only the labelings change* as a consequence of this, not the CFGs. The following lemma is immediate:

Lemma 5. *Let* $\mathbf{p}$, $\mathbf{q}$ *be two procedures such that* $\mathbf{body}(\mathbf{p})$ *and* $\mathbf{body}(\mathbf{q})$ *differ only in the annotated loop invariants. Then* $CFG(\mathbf{p}) = CFG(\mathbf{q})$.

An interactive loop can work as follows: suppose some (automated or user-guided) strategy is used on the initial program to export a set of edge conditions to the prover, and some of these are disproved. The user can then edit some of the annotations (formally modifying the program), which will cause the affected edge conditions to be recalculated according to definitions 4 and 9. At this point VCs can again be selected based on some strategy.

One of the most difficult tasks of program verification is identifying the appropriate invariant for each loop. Invariants may benefit from interactive editing, since forward propagation (used before the loop) and backward propagation (used after the loop) may help in identifying the invariant. Note that if an invariant is edited, this requires recalculating both labelings. But in any case, the program does not have to be parsed and the CFG reconstructed; only the modified annotation is parsed and the labelings partially recomputed by local propagation along the graph.

Consider for instance a procedure $\mathbf{p}$ such that $\mathbf{body}(\mathbf{p})$ is the block S_1 ; $\mathbf{while}\, b\, \mathbf{do}\, \{I\}\, S_w$; S_2. The user could choose to propagate the conditions $\mathbf{pre}(\mathbf{p})$ forward through S_1 and $\mathbf{post}(\mathbf{p})$ backward through S_2, and then examine the conditions of the edges adjacent to the loop's subgraph, $\mathsf{spost}(S_1, \mathbf{pre}(\mathbf{p})) \to I$ and $I \wedge \neg\, b \to \mathsf{wprec}(S_2, \mathbf{post}(\mathbf{p}))$, which must be both valid in addition to the invariant preservation condition. The invariant I could then be edited, and the validity of the three updated conditions checked interactively until an appropriate invariant was found.

Intermediate Conditions A feature that can increase even more the advantages of interactivity is the possibility to check conditions inserted at arbitrary points of the verification graph. An intermediate assertion must hold at the point where it is inserted, and provides information that can be used subsequently. Introducing assertions in a verification graph can be a great help when automation fails; they act as lemmas that are easy to prove but may then allow for more difficult VCs to be discharged automatically.

Suppose e is an edge with origin N and destination M in a verification graph with edge condition $P \to Q$. An intermediate assertion A can be created by removing e and inserting a new $\mathbf{assert}$ node O and two edges $e_1 : N \to O$ with edge condition $P \to A$, and $e_2 : O \to M$ with edge condition $A \to Q$. Such nodes are not covered by Lemma 3, thus Proposition 5 would now state that for every execution path ϕ crossing an $\mathbf{assert}$ node an adequate set of edges should contain two edges of ϕ, one before the node and another after the node.

7 Conclusions

We have studied the generation of verification conditions combining forward and backward propagation of assertions. For this we have introduced verification graphs and studied their properties. We have shown that our approach provides a general setting for error path discovery, VC splitting, and interactive verification, and it can be used to generate efficient VCs for passive programs.

One main challenge would be to produce a robust tool for intermediate code of a major verification platform, such as Why [11] or Boogie [2]. This would allow us to test the ideas with realistic code, since several verification tools for languages like C, Java, or Spec# are based on these platforms.

References

[1] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, first edition, March 2003.

[2] 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.

[3] Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT Softw. Eng. Notes*, 31(1):82–87, 2006.

[4] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS : construction and analysis of safe, secure, and interoperable smart devices*, volume 3362, pages 49–69. Springer, Berlin, ALLEMAGNE, March 2004.

[5] José Barros, Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based Slicing and Slice Graphs. In José Luis Fiadeiro and Stefania Gnesi, editors, *Proceedings of the eighth IEEE International Conference on Software Engineering and Formal Methods (SEFM'10)*, pages 93–102, 2010.

[6] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 323–335, London, UK, 1996. Springer-Verlag.

[7] Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theor. Comput. Sci.*, 173:49–87, February 1997.

[8] I. A. Browne, Zohar Manna, and Henny Sipma. Generalized temporal verification diagrams. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 484–498, London, UK, 1995. Springer-Verlag.

[9] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

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

[11] 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.

[12] 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.

[13] Robert Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, number 19 in Proceedings of Symposia in Applied Mathematics, pages 19–32. American Mathematical Society, 1967.

[14] Maria João Frade and Jorge Sousa Pinto. Verification Conditions for Source-level Imperative Programs. *Computer Science Review*, 2011. DOI: 10.1016/j.cosrev.2011.02.002.

[15] G. C. Gannod and B. H. C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, pages 166–, Washington, DC, USA, 1995. IEEE Computer Society.

[16] 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.

[17] Mike Gordon and Hélène Collavizza. Forward with Hoare. Draft dated July 12, 2010, 2010.

[18] Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest postcondition of unstructured programs. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, FTfJP '09, pages 6:1–6:7, New York, NY, USA, 2009. ACM.

[19] Robert M. Hierons, Mark Harman, Chris Fox, Lahcen Ouarbya, and Mohammed Daoudi. Conditioned slicing supports partition testing. *Softw. Test., Verif. Reliab.*, pages 23–28, 2002.

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

[21] Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *Comput. J.*, 38(2):131–141, 1995.

[22] Ivan Jager and David Brumley. Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, Carnegie Mellon University, 2010.

[23] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.

[24] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[25] K. Rustan M. Leino, Michal Moskal, and Wolfram Schulte. Verification condition splitting. Microsfot Research, 2008.

[26] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[27] Si Pan and R. Geoff Dromey. Using strongest postconditions to improve software quality. In *Software Quality and Productivity: Theory, practice and training*, pages 235–240, London, UK, UK, 1995. Chapman & Hall, Ltd.

[28] Corina S. Pasareanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11:339–353, October 2009.

A Proofs

This section is dedicated to the proof of Proposition 4. The proposition follows from Lemmas 7 and 8 below, as well as Proposition 3. We first prove an auxiliary result. The following lemma establishes a correspondence between the set of edge conditions, which are propagated locally through the graph, and the recursive structure of the underlying program.

Lemma 6. *Let $S = C_1, \ldots, C_n$. Then*

$$\mathsf{EC}\,(P, S, Q) \;=\; \bigcup_{k \in \{0 \ldots n\}} \{\mathsf{spost}^k(S, P) \rightarrow \mathsf{wprec}^{k+1}(S, Q)\} \;\cup\; \mathsf{EC}_k$$

where

$$\begin{array}{lcl}
\mathsf{EC}_k & = & \emptyset, \text{ if } k = 0 \text{ or } C_k \text{ is a } \mathbf{skip}\text{, assignment, or procedure call command} \\
\mathsf{EC}_k & = & \mathsf{EC}\left(\mathsf{spost}^{k-1}(S, P) \wedge b, S_t, \mathsf{wprec}^{k+1}(S, Q)\right) \cup \\
 & & \mathsf{EC}\left(\mathsf{spost}^{k-1}(S, P) \wedge \neg b, S_f, \mathsf{wprec}^{k+1}(S, Q)\right), \text{ if } C_k \text{ is } \mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f \\
\mathsf{EC}_k & = & \mathsf{EC}\,(I \wedge b, S, I)\,, \text{ if } C_k \text{ is } \mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S
\end{array}$$

Proof. By induction on the length n of S. If $n = 1$ then S is a command C. The set of edges of the graph $CFG(S)$ consists of two edges, e_{START} departing from $START$ with $wplb^Q(e_{START}) = \mathsf{wprec}(C, Q)$ and $splb^P(e_{START}) = P$, and e_{END} leading to END with $wplb^Q(e_{END}) = Q$ and $splb^P(e_{END}) = \mathsf{spost}(C, P)$, together with the edges of the CFGs of the subblocks of C. If C is one of **skip**, assignment, or procedure call, then there are no such subblocks and the lemma holds trivially.

If C is a conditional $\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f$ then $CFG(S)$ contains also the edges of $CFG(S_t)$ and $CFG(S_f)$; let e_{tSTART} / e_{tEND} and e_{fSTART} / e_{fEND} be the edges of each of these subgraphs connected respectively to $IN(C)$ / $OUT(C)$. Definition 4 prescribes that $splb^P(e_{tSTART}) = P \wedge b$, $wplb^Q(e_{tEND}) = Q$, $splb^P(e_{fSTART}) = P \wedge \neg b$, and $wplb^Q(e_{fEND}) = Q$. Thus the labeling of $CFG(S_t)$ and $CFG(S_f)$ by the previous definitions will produce exactly the sets of edge conditions $\mathsf{EC}\,(P \wedge b, S_t, Q)$ and $\mathsf{EC}\,(P \wedge \neg b, S_f, Q)$. A similar reasoning applies when C is $\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S_w$.

When S is the block $C_1\,;\,\ldots\,;\,C_n$ with $n > 1$, the result follows from a careful consideration of the edge conditions of subblocks of C_n, the induction hypothesis
$\mathsf{EC}\,(P, C_1\,;\,\ldots\,;\,C_{n-1}, \mathsf{wprec}(C_n, Q)) = \bigcup_{k \in \{0 \ldots n-1\}} \{\mathsf{spost}^k(S, P) \rightarrow \mathsf{wprec}^{k+1}(S, \mathsf{wprec}(C_n, Q))\} \;\cup\; \mathsf{EC}_k$, and the labels of the edge leading to END, with condition $\mathsf{spost}(S, P) \rightarrow Q$.

□

The next lemma states that the verification graph contains *all* the equivalent sets of verification conditions mentioned in Proposition 3.

Lemma 7. *Let P, Q be assertions, $S = C_1\,;\,\ldots\,;\,C_n$ a block of commands, and $k \in \{0, \ldots, n\}$. Then*

$$\mathsf{VCG}^k(P,\, S,\, Q) \subseteq \mathsf{EC}\,(P, S, Q)$$

Proof. By induction on the structure of S. If S is a single command, then $n = 1$ and k can only be 0 or 1; we just have to prove $\mathsf{VCG}(P,\, S,\, Q) \subseteq \mathsf{EC}\,(P, S, Q)$ and $\overline{\mathsf{VCG}}(P,\, S,\, Q) \subseteq \mathsf{EC}\,(P, S, Q)$.

For the base cases of **skip**, assignment, and procedure call, we have $\mathsf{VCG}(P,\, S,\, Q) = \{P \rightarrow \mathsf{wprec}(C, Q)\}$ and $\overline{\mathsf{VCG}}(P,\, S,\, Q) = \{\mathsf{spost}(C, P) \rightarrow Q\}$, both subsets of $\mathsf{EC}\,(P, S, Q)$ following Lemma 6.

If S is $\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f$, by Lemma 1 (1), $\mathsf{VCG}(P,\, S,\, Q) = \mathsf{VCG}(P \wedge b,\, S_t,\, Q) \cup \mathsf{VCG}(P \wedge \neg b,\, S_f,\, Q)$. We have by induction hypotheses $\mathsf{VCG}(P \wedge b,\, S_t,\, Q) \subseteq \mathsf{EC}\,(P \wedge b, S_t, Q)$ and $\mathsf{VCG}(P \wedge \neg b,\, S_f,\, Q) \subseteq \mathsf{EC}\,(P \wedge \neg b, S_f, Q)$, and by Lemma 6, with $n = 1$, $\mathsf{EC}\,(P \wedge b, S_t, Q) \subseteq \mathsf{EC}\,(P, S, Q)$ and also
$\mathsf{EC}\,(P \wedge \neg b, S_f, Q) \subseteq \mathsf{EC}\,(P, S, Q)$. This is proved similarly for $\overline{\mathsf{VCG}}(P,\, S,\, Q)$.

If S is $\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S_w$, by Lemma 1 (2) $\mathsf{VCG}(P,\, S,\, Q) = \{P \rightarrow \mathsf{wprec}(S, Q), \mathsf{spost}(S, P) \rightarrow Q\} \cup \mathsf{VCG}(I \wedge b,\, S_w,\, I)$. By induction hypothesis and Lemma 6 (again with $n = 1$), $\mathsf{VCG}(I \wedge b,\, S_w,\, I) \subseteq \mathsf{EC}\,(I \wedge b, S_w, I) \subseteq \mathsf{EC}\,(P, S, Q)$; the latter lemma also implies $\{P \rightarrow \mathsf{wprec}(S, Q), \mathsf{spost}(S, P) \rightarrow Q\} \subseteq \mathsf{EC}\,(P, S, Q)$. Again $\overline{\mathsf{VCG}}(P,\, S,\, Q) \subseteq \mathsf{EC}\,(P, S, Q)$ is proved similarly.

Finally, if S is a block with $n > 1$, we first note that by Lemma 1 (3)

$$\begin{array}{rl}
\mathsf{VCG}^k(P,\, S,\, Q) = & \overline{\mathsf{VCG}}(P,\, C_1\,;\,\ldots\,;\,C_k,\, \mathsf{wprec}^{k+1}(S, Q)) \\
 & \cup\, \mathsf{VCG}(\mathsf{spost}^k(S, P),\, C_{k+1}\,;\,\ldots\,;\,C_n,\, Q)
\end{array}$$

And we have the following induction hypotheses:

$$\overline{\mathsf{VCG}}(P,\, C_1\,;\,\ldots\,;\,C_k,\,\mathsf{wprec}^{k+1}(S,Q)) \subseteq \mathsf{EC}\left(P, C_1\,;\,\ldots\,;\,C_k, \mathsf{wprec}^{k+1}(S,Q)\right)$$
$$\mathsf{VCG}(\mathsf{spost}^k(S,P),\, C_{k+1}\,;\,\ldots\,;\,C_n,\,Q) \subseteq \mathsf{EC}\left(\mathsf{spost}^k(S,P), C_{k+1}\,;\,\ldots\,;\,C_n, Q\right)$$

Now it suffices to note that by definition of $CFG(S)$ and Lemma 6, one has

$$\begin{aligned}\mathsf{EC}\,(P,S,Q) = \mathsf{EC}\left(P, C_1\,;\,\ldots\,;\,C_k, \mathsf{wprec}^{k+1}(S,Q)\right)\\ \cup\, \mathsf{EC}\left(\mathsf{spost}^k(S,P), C_{k+1}\,;\,\ldots\,;\,C_n, Q\right)\end{aligned}$$

□

We remark that the reverse of the previous lemma does not hold: the edge conditions of the labeled CFG may well contain more assertions than *any* $\mathsf{VCG}^k(P,\, S,\, Q)$. Instead we prove the following:

Lemma 8. *Let P, Q be assertions, $S = C_1\,;\,\ldots\,;\,C_n$ a block of commands such that $\models \mathsf{VCG}(P,\, S,\, Q)$; then $\models \mathsf{EC}\,(P,S,Q)$.*

Proof. First note that by Proposition 3 we have that $\models \mathsf{VCG}^k(P,\, S,\, Q)$ for all $k \in \{0,\ldots,n\}$. The proof proceeds by induction on the structure of S. If S is a single command C, then $n = 1$ and k can only be 0 or 1; if C is atomic we note that by Lemma 6 $\mathsf{EC}\,(P,S,Q) = \{\mathsf{spost}^0(S,P) \to \mathsf{wprec}^1(S,Q), \mathsf{spost}^1(S,P) \to \mathsf{wprec}^2(S,Q)\}$, and according to Definition 2, we have $\mathsf{VCG}^0(P,\, S,\, Q) \ni \{\mathsf{spost}^0(S,P) \to \mathsf{wprec}^1(S,Q)\}$ and $\mathsf{VCG}^1(P,\, S,\, Q) \ni \{\mathsf{spost}^1(S,P) \to \mathsf{wprec}^2(S,Q)\}$.

Otherwise, if C is $\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f$, the set of edge conditions $\mathsf{EC}\,(P,S,Q)$ contains also $\mathsf{EC}_1 = \mathsf{EC}\left(\mathsf{spost}^0(S,P) \wedge b, S_t, \mathsf{wprec}^2(S,Q)\right) \cup \mathsf{EC}\left(\mathsf{spost}^0(S,P) \wedge \neg b, S_f, \mathsf{wprec}^2(S,Q)\right)$. It then suffices to apply Lemma 1 (1) and induction hypotheses: $\models \mathsf{VCG}(P \wedge b,\, S_t,\, Q)$ implies $\models \mathsf{EC}\,(P \wedge b, S_t, Q)$ and $\models \mathsf{VCG}(P \wedge \neg b,\, S_f,\, Q)$ implies $\models \mathsf{EC}\,(P \wedge \neg b, S_f, Q)$.

If S is $\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S_w$, then the set of edge conditions $\mathsf{EC}\,(P,S,Q)$ contains $\mathsf{EC}_1 = \mathsf{EC}\,(I \wedge b, S_w, I)$, and by induction hypothesis $\models \mathsf{VCG}(I \wedge b,\, S_w,\, I)$ implies $\models \mathsf{EC}\,(I \wedge b, S_w, I)$. Lemma 1 (2) concludes the proof.

If S is a block with $n > 1$, recall from the proof of the Lemma 7 that

$$\begin{aligned}\mathsf{EC}\,(P,S,Q) = \mathsf{EC}\left(P, C_1\,;\,\ldots\,;\,C_k, \mathsf{wprec}^{k+1}(S,Q)\right)\\ \cup\, \mathsf{EC}\left(\mathsf{spost}^k(S,P), C_{k+1}\,;\,\ldots\,;\,C_n, Q\right)\end{aligned}$$

The ind. hypotheses $\models \mathsf{VCG}(P,\, C_1\,;\,\ldots\,;\,C_k,\, \mathsf{wprec}^{k+1}(S,Q))$ implies $\models \mathsf{EC}\left(P, C_1\,;\,\ldots\,;\,C_k, \mathsf{wprec}^{k+1}(S,Q)\right)$ and $\models \mathsf{VCG}(\mathsf{spost}^k(S,P),\, C_{k+1}\,;\,\ldots\,;\,C_n,\, Q)$ implies $\models \mathsf{EC}\left(\mathsf{spost}^k(S,P), C_{k+1}\,;\,\ldots\,;\,C_n, Q\right)$ and Proposition 3 mean that it suffices to establish the validity of $\overline{\mathsf{VCG}}(P,\, C_1\,;\,\ldots\,;\,C_k,\, \mathsf{wprec}^{k+1}(S,Q))$ and of $\mathsf{VCG}(\mathsf{spost}^k(S,P),\, C_{k+1}\,;\,\ldots\,;\,C_n,\, Q)$, which follows from Lemma 1 (3).

□

```
1  public class UKTaxesCalculation
2  {
3      public int age, income;
4      public int personal, t;
5
6      /*@ requires (age >= 65);
7          ensures (personal > 5750);
8       @*/
9      public void TaxesCalculation()
10     {
11         if (age >= 75) { personal = 5980; }
12         else if (age >= 65) { personal = 5720; }
13              else { personal = 4335; }
14
15         if ((age >= 65) && (income > 16800))
16         {
17             t = personal - ((income-16800)/2);
18             if (t > 4335) { personal = t + 2000; }
19             else { personal = 4335; }
20         }
21     }
22 }
```

Listing 1: Class TaxesCalculation

B Interactive Verification with GamaAnimator

In this Appendix we briefly show how our prototype tool GamaAnimator can be used as an interactive VCGen (for a subset of a popular object-oriented language). The tool is part of the GamaSlicer suite, and has the following functionality:

- Visualization of verification graphs and user-guided generation of verification conditions by interacting with the graphs;
- Propagation of the valid and invalid status of edge conditions, through the use of a color code, allowing to identify error paths.

Color is used in the following way: edges with invalid conditions are shown in *red*; edges with valid conditions are shown in *green*; other edges are shown in *black*. Colors are propagated along the graph following Lemma 3, the idea being that paths from $START$ to END consisting only of edges displayed in red are error paths, that violate the postcondition.

- The green color propagates freely through atomic command nodes in either direction. In branching nodes, it propagates from outside the conditional command into both branches; but it only propagates outwards, in both directions, when both branch edges are green.
- The red color also propagates freely across atomic command nodes. In branching nodes, it does *not* propagate from outside the conditional command into the branches (since it cannot be guessed which branch – or branches – have an invalid VC), but it does propagate *over* the entire conditional subgraph (if the edge leading to an $\mathbf{if}(b)$ node is red, then so is the edge with origin in the corresponding **fi** node, and vice-versa); it propagates outwards from any branch (without requiring the other branch to also be red) in both directions.
- Every $\mathbf{do}(b)\{I\}$ and $\mathbf{od}\{I\}$ node blocks the propagation of both the red and green colors, since no equivalence exists between the edge conditions generated from a loop.

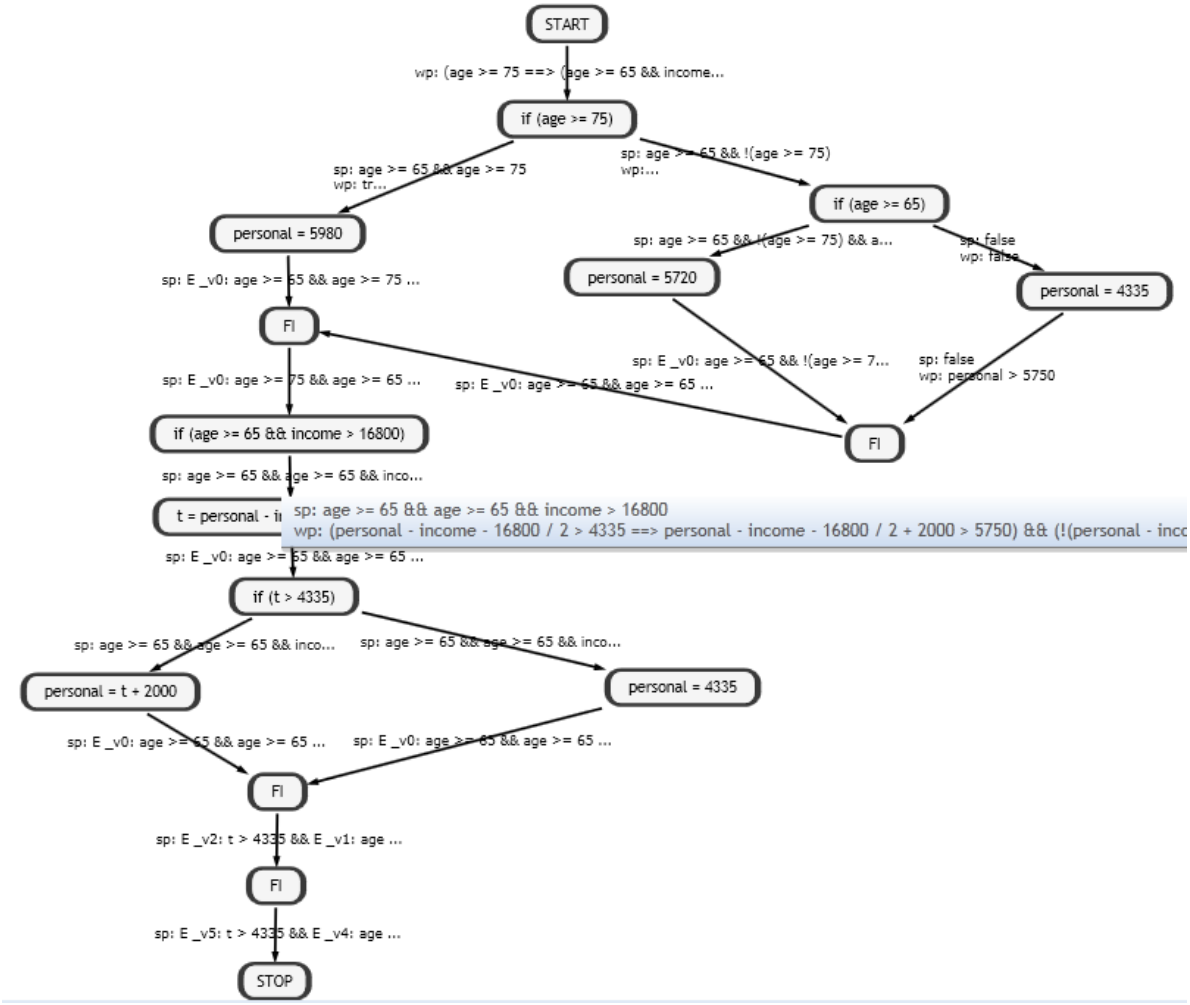
Nodes in the graph will also be displayed in color, as follows: a node is shown in green if *all* its incoming and outgoing edges are green; it is shown in red if *at least one* of its incoming and outgoing edges is red; it is shown in black otherwise (i.e. no red edges and at least one black edge).

Note that the verification is finished when the edge with origin in $START$ or the edge with destination END are green or red, signaling a correct or incorrect program. This does not imply that every edge in the graph is either green or red, which may require more work in order to identify specific error paths.

To illustrate the use of the tool, consider the method `TaxesCalculation` in Listing 1, a fragment of a program used to calculate income taxes in the United Kingdom,[1] with contract $(age \geq 65, personal > 5750)$. The (invalid) verification condition generated by backward propagation is shown in Figure 8 (bottom). It is not straightforward from the observation of this VC to understand which statements lead to the violation of the contract.

The verification graph of the method is visualized by GamaAnimator as shown in Figure 8. Each edge is labeled with assertions propagated forward and backward. To avoid big labels in the graph, we simplify the

[1] The complete source code of this program can be found in [19], where it is used as a benchmark for slicing algorithms based on assertions.




```
:formula (not (and (implies (>= ?age 75 ) (and (implies (and (>= ?age 65 ) (> ?income 16800 )
     ) (and (implies (> (- 5980 (div (- ?income 16800 ) 2 ) ) 4335 ) (> (- 5980 (div
     (- ?income 16800 ) 2 ) ) 5750 ) ) (implies (not (> (- 5980 (div (- ?income 16800
     ) 2 ) ) 4335 ) ) (> 4335 5750 ) ) ) ) (implies (not (and (>= ?age 65 ) (> ?income
     16800 ) ) ) (> 5980 5750 ) ) ) ) (implies (not (>= ?age 75 ) ) (and (implies (>=
     ?age 65 ) (and (implies (and (>= ?age 65 ) (> ?income 16800 ) ) (and (implies (>
     (- 5720 (div (- ?income 16800 ) 2 ) ) 4335 ) (> (- 5720 (div (- ?income 16800 )
     2 ) ) 5750 ) ) (implies (not (> (- 5720 (div (- ?income 16800 ) 2 ) ) 4335 ) )
     (> 4335 5750 ) ) ) ) (implies (not (and (>= ?age 65 ) (> ?income 16800 ) ) ) (>
     5720 5750 ) ) ) ) (implies (not (>= ?age 65 ) ) (and (implies (and (>= ?age 65
     ) (> ?income 16800 ) ) (and (implies (> (- 4335 (div (- ?income 16800 ) 2 ) ) 4335
     ) (> (- 4335 (div (- ?income 16800 ) 2 ) ) 5750 ) ) (implies (not (> (- 4335 (div
     (- ?income 16800 ) 2 ) ) 4335 ) ) (> 4335 5750 ) ) ) ) (implies (not (and (>= ?age
     65 ) (> ?income 16800 ) ) ) (> 4335 5750 ) ) ) ) ) ) )
```

Figure 8: Verification graph of the `TaxesCalculation` Method and condition of the edge with origin in the $START$ node

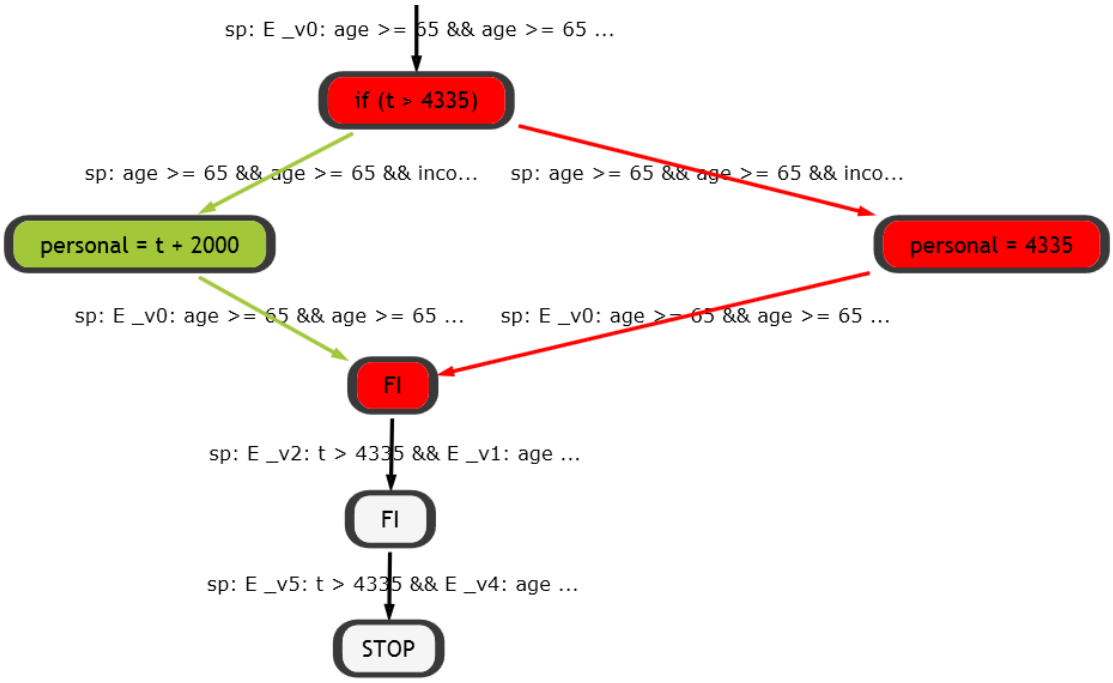


Figure 9: Verification of edge conditions of `TaxesCalculation` (1)

labels of each edge whenever possible. For this simplification, we take advantage of an internal simplifier of the Z3 solver. If the simplified expression is still big enough to unbalance the layout of the graph, then a part of the edge is replaced by dots (...) and a tooltip is displayed when the mouse is placed over the edge, as shown in the figure. The user can now select edge conditions to be checked. When an edge is clicked on, the corresponding edge condition is sent to the prover. The tool keeps track of the current list of conditions being checked, and a history of conditions previously considered.

Let us zoom in on the subgraph corresponding to the second conditional in the method, and select the last edge inside the *then* branch. The following edge condition is sent to the prover and identified as *valid*:[2]

$$\exists v0.\, age \geq 65 \wedge income > 16800 \wedge t > 4335 \wedge personal = t + 2000 \rightarrow personal > 5750$$

Thus, the edges inside the branch and the assignment instruction are shown in green. This means that the statements inside the *then* branch are not the cause of the problem. If we now pick the last edge inside the *else* branch, the following condition is sent to the prover and identified as not valid:

$$\exists v0.\, age \geq 65 \wedge income > 16800 \wedge t > 4335 \wedge personal \neq t + 2000 \rightarrow personal > 5750$$

At this point, we know that this branch is part of an error path. The edges inside the branch, the assignment node, and the **if**$(t > 4335)$ and **fi** nodes all become red (Figure 9).

[2]In fact it is the *negation* of this formula that is sent to an SMT solver, which returns *unsat*, meaning that the original formula is valid.