Under consideration for publication in Formal Aspects of Computing

Assertion-based Slicing and Slice Graphs

José Bernardo Barros, Daniela da Cruz, Pedro Rangel Henriques and Jorge Sousa Pinto

Departamento de Informática / CCTC
Universidade do Minho
Braga, Portugal

Abstract. This paper revisits the idea of slicing programs based on their axiomatic semantics, rather than using criteria based on control/data dependencies. We show how the forward propagation of preconditions and the backward propagation of postconditions can be combined in a new slicing algorithm that is more precise than the existing specification-based algorithms. The algorithm is based on (i) a precise test for removable statements, and (ii) the construction of a *slice graph*, a program control flow graph extended with semantic labels and additional edges that "short-circuit" removable commands. It improves on previous approaches in two aspects: it does not fail to identify removable commands; and it produces the smallest possible slice that can be obtained (in a sense that will be made precise). Iteration is handled through the use of loop invariants and variants to ensure termination.

The paper also discusses in detail applications of these forms of slicing, including the elimination of (conditionally) unreachable and dead code, and compares them to other related notions.

Keywords: Program slicing; program analysis; verification conditions; control flow graphs.

1. Introduction

Program slicing [Wei81] is a well-established activity in software engineering. It plays an important role in program comprehension, since it allows software engineers to focus on the relevant portions of code (with respect to a given criterion). The basic idea is to isolate a subset of program statements that

- either directly or indirectly contribute to the values of a set of variables at a given program location, or
- are influenced by the values of a given set of variables.

Other statements are considered extraneous with respect to the given criterion and can be removed, enabling engineers to concentrate on the analysis of just the relevant ones. The first approach corresponds to *backward* forms of slicing, whereas the second corresponds to *forward* slicing.

Correspondence and offprint requests to: Daniela da Cruz, Departamento de Informática, Universidade do Minho, Campus de Gualtar, 4710-057 Braga, Portugal. e-mail: danieladacruz@di.uminho.pt

Work in this area has focused on the development of progressively more effective, useful, and powerful slicing techniques, and has led to the use of these techniques in many application areas including program debugging, software maintenance, software reuse, and so on. See for instance [XQZ+05] for a fairly recent survey of the area.

Program verification is an apparently unrelated activity whose goal is to establish that a program performs according to some intended specification. Typically, what is meant by this is that the input/output behaviour of the implementation matches that of the specification (this is usually called the *functional* behaviour of the program), and moreover the program does not 'go wrong', for instance no errors occur during evaluation of expressions (the so-called *safety* behaviour). Modern program verification systems are based on algorithms that examine a program and generate a set of *verification conditions* that are sent to an external theorem prover for checking. If all the conditions generated from a program can be proved, then the program is guaranteed to be correct with respect to the specification.

In recent years program verification has been closely linked with the so-called *Design by Contract* (DbC) approach to software development [Mey92], which facilitates modular verification and certified code reuse. The contract for a software component can be regarded as a form of enriched software documentation that fully specifies the behavior of that component. In terms of verification terminology, a contract for a component is simply a pair consisting of a precondition and a postcondition. It certifies the results that can be expected after execution of the component, but it also constrains the input values of the component. The development and broad adoption of annotation languages for the major programming languages reinforces the importance of using DbC principles in program development. These include for instance the Java Modeling Language (JML) [BCC+05]; Spec# [BRLS04], a formal language for C# API contracts; and the ANSI/ISO C Specification Language (ACSL) [BCF+10].

One point of contact that has been identified between slicing and verification is that traditional dependency-based slicing, applied a priori, facilitates the verification of large programs. In this paper we explore the idea that it makes sense to slice programs based on semantic, rather than syntactic, criteria – the contracts used in DbC and program verification are excellent candidates for such criteria.

A typical example of a situation in which one could wish to calculate the slice of a program based on a specification is the reuse of annotated code. Suppose one is interested in reusing a module whose advertised contract consists of precondition P and postcondition Q, in situations in which a stronger precondition P' is known to hold, or else the desired postcondition Q' is weaker than the specified Q. Then from a software engineering perspective it would be desirable to eliminate, at source-level, the code that may be extraneous with respect to the specification (P', Q').

We use here the expression "assertion-based slicing" to refer to slicing methods based on the axiomatic semantics of programs, taking as criteria assertions (preconditions and/or postconditions) annotated in the programs. This includes *precondition-based* slicing, *postcondition-based* slicing, and *specification-based* slicing. The latter expression has been used in previous work when both a precondition *and* a postcondition (i.e. a specification) are given as criteria. Assertion-based slicing is more powerful and flexible than syntactic slicing, since the criteria can be as expressive as any set of first-order formulas on the initial and final states of the program. One of the first forms of slicing based on program semantics was *conditioned slicing* [CCL98], a form of forward slicing. This was shown to subsume both static and dynamic notions of dependency-based slicing, since the initial state of execution is constrained by a first-order formula that can be used to restrict the set of admissible initial states to exactly one (corresponding to dynamic slicing), or simply to identify a relevant subset of the state to be used as slicing criterion (as in static slicing). The same applies to backward slicing: using a postcondition as slicing criterion instead of a set of variables is clearly more expressive. Naturally, this expressiveness comes at a cost, since semantic forms of slicing are harder to compute.

Although the basic ideas have been published for over 10 years now, assertion-based slicing is still not very popular – in particular we are not aware of working tools that implement the ideas. The widespread usage of code annotations as explained above is however an additional argument for promoting it. This work is part of an effort to construct a complete toolset for assertion-based slicing.

The paper reviews (and clarifies aspects of) previous work in this area, sets a basis for slicing programs annotated with loop invariants and variants, and studies properties and applications of such slices. We introduce new ideas which allow us to develop an algorithm for specification-based slicing that improves on previous algorithms in two aspects: the identification of sequences of statements that can be safely removed from a program (without modifying its semantics), and the selection of the biggest set of such sequences.

Note that removable sequences may overlap, so this is not a trivial problem. We solve it by introducing a notion of *slice graph*, which contains as subgraphs the control flow graph of every slice of the program. This allows us to define a slicing algorithm that can be applied to calculate precondition-, postcondition-, and specification-based slices, but we concentrate on the latter, since the first two are particular cases.

We claim that our algorithm produces minimal slices. Note that the algorithm is *optimal in a relative sense*, since the test for removable subprograms involves first-order formulas whose validity must be established externally by some proof tool. Undecidability of first-order logic destroys any hope of being able to identify every removable subprogram automatically, since some valid formulas may not be proved.

This paper extends [BdCHP10] in a number of ways. An appropriate treatment of loops is given, and a new notion of termination-sensitive slicing is introduced. More examples have been added, and proofs, related work, slicing algorithms, and applications (in particular in the elimination of redundant code) are discussed in much more detail.

Structure of the Paper. Section 2 introduces the simple language considered in the paper, and the definitions of verification conditions, both based on weak precondition and strong postcondition calculations. In Section 3 we review the previous work in this area, including a discussion of aspects of the extant algorithms regarding their precision and minimality of the calculated slices. Section 4 then discusses in detail applications of assertion-based slicing, illustrated by a number of examples. Sections 5, 6 and 7 contain the main technical contributions of the paper: we first study properties of specification-based slicing and propose a precise test for identifying removable blocks of code, as well as a principle for slicing subprograms of a program. Later we introduce the setting for a graph-based algorithm that computes minimal slices of a program with respect to a given specification. We conclude the paper in Section 8.

2. Foundations

A Simple Imperative Language. We will illustrate our ideas with programs of a core imperative language. Its syntax is given in Figure 1 (the remaining contents of the figure are explained below). Programs are non-empty sequences of commands. Commands may in turn (in the case of conditional and loop) contain subprograms. We omit here the operational semantics of the language, and give instead an axiomatic semantics in the form of a *Verification Conditions Generator* (VCGen), which will now be explained.

We remark that the choice of language is not important; the only crucial requirements are the existence of an axiomatic semantics, and the availability of a proof tool capable of reasoning about the data structures that are present in the language. The difficulties involved in extending the ideas presented here to realistic languages (for instance languages with pointers and dynamic data structures) have to do with the treatment of verification conditions only.

Weakest Preconditions and Strongest Postconditions. One way to obtain a verification condition for a program S (i.e. a formula whose validity implies the partial correctness[1] of S with respect to a specification consisting of precondition P and postcondition Q) is to use Dijkstra's *weakest liberal precondition* [Dij76] predicate transformer: $wlp.S.Q$ designates the weakest precondition that will lead to Q being true in the final state, *if the execution of* S *terminates*. The verification condition for S to meet its specification can then be written as $P \rightarrow wlp.S.Q$. In this paper we assume that every loop is annotated with a user-provided *invariant*; in the presence of a loop invariant, the different required conditions (the invariant is initially true, it is preserved by loop iterations, and together with the condition for exiting the loop it is stronger that the desired postcondition) are combined in a single formula to give a possible precondition of each loop. This requires the use of universal quantifiers over state variables, to isolate the different conditions.

In this paper we use a different approach that produces a set of independent verification conditions, dispensing with the introduction of quantifiers. This approach requires calculating a notion of precondition that is related to *wlp*, but differs in that the precondition of a loop is simply defined as being simply its invariant, regardless of whether termination is guaranteed or not. The figure shows the definition of the function wprec corresponding to this notion. Throughout the paper whenever we refer to a weak precondition of a program, we mean a condition calculated by this function.

[1] Unlike *total correctness*, the notion of *partial correctness* does not require the program to terminate.

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

$$
\begin{array}{rcl}
\mathsf{wprec}(\textbf{skip}, Q) & = & Q \\
\mathsf{wprec}(x := e, Q) & = & Q[e/x] \\
\mathsf{wprec}(\textbf{if}\ b\ \textbf{then}\ S_t\ \textbf{else}\ S_f, Q) & = & (b \rightarrow \mathsf{wprec}(S_t, Q)) \wedge\ (\neg b \rightarrow \mathsf{wprec}(S_f, Q)) \\
\mathsf{wprec}(\textbf{while}\ b\ \textbf{do}\ \{I\}\ S, Q) & = & I \\
\mathsf{wprec}(C; S, Q) & = & \mathsf{wprec}(C, \mathsf{wprec}(S, Q))
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{VC}^w(\textbf{skip}, Q) & = & \emptyset \\
\mathsf{VC}^w(x := e, Q) & = & \emptyset \\
\mathsf{VC}^w(\textbf{if}\ b\ \textbf{then}\ S_t\ \textbf{else}\ S_f, Q) & = & \mathsf{VC}^w(S_t, Q)\ \cup\ \mathsf{VC}^w(S_f, Q) \\
\mathsf{VC}^w(\textbf{while}\ b\ \textbf{do}\ \{I\}\ S, Q) & = & \{I \wedge b \rightarrow \mathsf{wprec}(S, I), I \wedge \neg b \rightarrow Q\}\ \cup\ \mathsf{VC}^w(S, I) \\
\mathsf{VC}^w(C; S, Q) & = & \mathsf{VC}^w(C, \mathsf{wprec}(S, Q))\ \cup\ \mathsf{VC}^w(S, Q)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{spost}(\textbf{skip}, P) & = & P \\
\mathsf{spost}(x := e, P) & = & \exists\, v.\, P[v/x] \wedge x = e[v/x] \\
\mathsf{spost}(\textbf{if}\ b\ \textbf{then}\ S_t\ \textbf{else}\ S_f, P) & = & \mathsf{spost}(S_t, b \wedge P) \vee\ \mathsf{spost}(S_f,\ \neg b \wedge P) \\
\mathsf{spost}(\textbf{while}\ b\ \textbf{do}\ \{I\}\ S, P) & = & I \wedge \neg b \\
\mathsf{spost}(C; S, P) & = & \mathsf{spost}(S, \mathsf{spost}(C, P))
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{VC}^s(\textbf{skip}, P) & = & \emptyset \\
\mathsf{VC}^s(x := e, P) & = & \emptyset \\
\mathsf{VC}^s(\textbf{if}\ b\ \textbf{then}\ S_t\ \textbf{else}\ S_f, P) & = & \mathsf{VC}^s(S_t, P)\ \cup\ \mathsf{VC}^s(S_f, P) \\
\mathsf{VC}^s(\textbf{while}\ b\ \textbf{do}\ \{I\}\ S, P) & = & \{P \rightarrow I, \mathsf{spost}(S, I \wedge b) \rightarrow I\}\ \cup\ \mathsf{VC}^s(S, I \wedge b) \\
\mathsf{VC}^s(C; S, P) & = & \mathsf{VC}^s(C, P)\ \cup\ \mathsf{VC}^s(S, \mathsf{spost}(C, P))
\end{array}
$$

Fig. 1. Language syntax and verification conditions. $Q[e/x]$ denotes the substitutions of e for x in Q; I is a loop invariant

The figure also contains the definition of the function spost, corresponding to the symmetric notion of a *strong postcondition* that will be true of the final state of the program S when its execution starts in a state satisfying P. The syntax of assertions (used as preconditions, postconditions, and loop invariants), also given in the figure, is obtained as an extension of boolean expressions with implication and first-order quantification.

Verification Conditions for Partial Correctness. An alternative way to calculate verification conditions is based on Hoare logic [Hoa69]. In this approach the verification conditions required for the partial

correctness of the program S with respect to specification (P, Q) are the side conditions of a derivation (or proof tree) of the logic. If the verification conditions are all valid, then it is possible to construct a derivation with the Hoare triple $\{P\}\, S\, \{Q\}$ as conclusion, in which case S is (partially) correct.

The derivations with a given conclusion are not unique; although they do not need to be explicitly constructed in order for the side conditions to be obtained, some strategy is still necessary to direct the process. In this paper we will use two such strategies, based on weak preconditions and on strong postconditions respectively. Technically, these strategies are responsible for selecting intermediate conditions for the *sequence* rule of Hoare logic: when considering a derivation for the triple $\{P\}\, S_1\,;\, S_2\, \{Q\}$, this rule states that two derivations should be recursively considered, for the triples $\{P\}\, S_1\, \{R\}$ and $\{R\}\, S_2\, \{Q\}$ for some condition R. Our first strategy sets R to be $\mathsf{wprec}(S_2, Q)$; the second strategy sets R to be $\mathsf{spost}(S_1, P)$.

Each of these strategies results in a different set of verification conditions, defined as follows.

$$\mathsf{VCG}^w(P,\, S,\, Q) = \{P \rightarrow \mathsf{wprec}(S, Q)\} \;\cup\; \mathsf{VC}^w(S, Q)$$

and

$$\mathsf{VCG}^s(P,\, S,\, Q) = \mathsf{VC}^s(S, P) \;\cup\; \{\mathsf{spost}(S, P) \rightarrow Q\}$$

where the functions VC^w and VC^s are also defined in the figure. These auxiliary functions are responsible for traversing the implicit derivations and collecting the side conditions along the way. The traversals are based uniquely on one of the conditions given in the specification (the postcondition and precondition respectively); the additional formula $P \rightarrow \mathsf{wprec}(S, Q)$ (resp. $\mathsf{spost}(S, P) \rightarrow Q$) added to this set is the principal verification condition of the program, stating that the specification's precondition is stronger than the calculated precondition (resp. the specification's postcondition is weaker than the calculated postcondition). For programs not containing loops, this is the single verification condition.

The generation of verification conditions should of course be *sound* with respect to the operational semantics of the language: if they are all valid then this should constitute a guarantee that the program is indeed correct with respect to its specification (P, Q):

If either $\models \mathsf{VCG}^w(P,\, S,\, Q)$ *or* $\models \mathsf{VCG}^s(P,\, S,\, Q)$ *and* S *is run in a state that satisfies* P*, then if* S *terminates the final state satisfies* Q*.*

This is easy to prove for such a simple language, with respect to a standard evaluation semantics. The reader is directed to [FP11] for a proof, and also for more details on verification conditions and their relation to Hoare logic and Dijkstra's predicate transformers.

We can state a correspondence result between both strategies as follows. Let $\models \mathcal{A}$, with $\mathcal{A}$ a set of first-order formulas, denote the fact that $\models A_i$ for every $A_i \in \mathcal{A}$ (set union will be denoted by commas).

Lemma 1. For every precondition P, postcondition Q, and program S,

$$\models \mathsf{VCG}^w(P,\, S,\, Q) \quad \text{iff} \quad \models \mathsf{VCG}^s(P,\, S,\, Q)$$

Proof. By induction on the structure of S. For the case where S is **while** b **do** $\{I\}\, S_b$ the following is used as induction hypothesis: $\models \mathsf{VCG}^w(I \wedge b,\, S_b,\, I)$ iff $\models \mathsf{VCG}^s(I \wedge b,\, S_b,\, I)$. □

Since our language has integer variables only, verifying the correctness of programs can be achieved by applying the VCGen and exporting the resulting proof obligations to a proof tool capable of reasoning with integer arithmetics. The result is a framework for the verification of programs with annotated loop invariants.

Observe that this is not a fully automated method since it requires users to provide the annotations. Furthermore, undecidability of first-order logic means that interactive proof is often necessary, but it must also be noted that the power of automatic proof has progressed significantly in recent years. Real-language implementations of many standard algorithms can now be proved fully automatically, which is certainly a great advance with respect to what could be achieved, say, ten years ago. Recent approaches build in particular on advances in SMT solvers (that combine useful programming theories), and also on combinations of automatic provers (for the easy proofs) and interactive proof assistants (for the hard parts).

In the scope of program verification, failure of automatic proof does not mean a program is not correct, it just means that interactive proof should be used instead to clarify whether a given proof obligation is indeed invalid or not. In the scope of the slicing techniques considered in this paper, proof obligations are generated to authorize the removal of a given set of statements. Slicing should be conservative, so failure of

an attempt to discharge a particular obligation implies that the set of statements being considered should not be removed.

Verification Conditions for Total Correctness. In a total correctness setting the verification conditions are further required to guarantee termination of programs. In our language no procedures or functions are present and expression evaluation always terminates, so what is required is that every loop in a given program terminates. For this we require each loop to contain an additional annotation, an integer expression e_v called a *loop variant*:

$$\textbf{Comm} \quad \ni \quad C \quad ::= \ldots \mid \textbf{while}\ b\ \textbf{do}\ \{A, e_v\}\ S$$

If for every loop in the program the value of the respective variant is initially non-negative and strictly decreases with each iteration while remaining nonnegative until the last iteration, the program is guaranteed to terminate. The VCGens of Figure 1 can be extended to cope with total correctness by simply modifying the verification conditions of loops. The function VC_t^w (resp. VC_t^s) has the same definition as VC^w (resp. VC^s) except for the case of loops, which is given as follows for a loop annotated with invariant I and variant e_v:

$$\begin{aligned}\mathsf{VC}_t^w(\textbf{while}\ b\ \textbf{do}\ \{I, e_v\}\ S, Q) = \ & \{I \wedge b \rightarrow e_v \geq 0,\ I \wedge b \wedge e_v = x_0 \rightarrow \mathsf{wprec}(S, I \wedge e_v < x_0),\ I \wedge \neg b \rightarrow Q\} \\ & \cup\ \mathsf{VC}_t^w(S, I \wedge e_v < x_0)\end{aligned}$$

$$\begin{aligned}\mathsf{VC}_t^s(\textbf{while}\ b\ \textbf{do}\ \{I, e_v\}\ S, P) = \ & \{I \wedge b \rightarrow e_v \geq 0,\ P \rightarrow I,\ \mathsf{spost}(S, I \wedge b \wedge e_v = x_0) \rightarrow I \wedge e_v < x_0\} \\ & \cup\ \mathsf{VC}_t^s(S, I \wedge b \wedge e_v = x_0)\end{aligned}$$

Note that the weak precondition and strong postcondition functions wprec and spost are still defined as before. Note also the use of an auxiliary variable x_0 to store the initial value of the variant (regarding an arbitrary loop iteration), which then allows us to force the postcondition $e_v < x_0$.[2] Now we let

$$\mathsf{VCG}_t^w(P, S, Q) = \{P \rightarrow \mathsf{wprec}(S, Q)\}\ \cup\ \mathsf{VC}_t^w(S, Q)$$

and

$$\mathsf{VCG}_t^s(P, S, Q) = \mathsf{VC}_t^s(S, P)\ \cup\ \{\mathsf{spost}(S, P) \rightarrow Q\}$$

The resulting VCGens are sound with respect to total correctness, i.e.

If either $\models \mathsf{VCG}_t^w(P, S, Q)$ *or* $\models \mathsf{VCG}_t^s(P, S, Q)$ *and* S *is executed in a state that satisfies* P*, then* S *terminates, and moreover the final state satisfies* Q*.*

Note that it is immediate from the definitions of VC_t^w and VC^w (resp. VC_t^s and VC^s) that

$$\models \mathsf{VCG}_t^w(P, S, Q) \quad \text{implies} \quad \models \mathsf{VCG}^w(P, S, Q), \quad \text{and}$$
$$\models \mathsf{VCG}_t^s(P, S, Q) \quad \text{implies} \quad \models \mathsf{VCG}^s(P, S, Q)$$

which is in accordance with the fact that total correctness is a stronger notion than partial correctness. In fact, in practice the total correctness of a program is often established by first proving its partial correctness and then additionally checking that it terminates on initial states satisfying the precondition.

Finally, the following lemma states that the weak precondition and the strong postcondition strategies are equivalent for calculating total correctness verification conditions:

Lemma 2. For every precondition P, postcondition Q, and program S,

$$\models \mathsf{VCG}_t^w(P, S, Q) \quad \text{iff} \quad \models \mathsf{VCG}_t^s(P, S, Q)$$

Proof. By induction on the structure of S. For the case where S is $\textbf{while}\ b\ \textbf{do}\ \{I, e_v\}\ S_b$, the following is used as induction hypothesis: $\models \mathsf{VCG}_t^w(I \wedge b \wedge e_v = x_0, S_b, I \wedge e_v < x_0)$ iff $\models \mathsf{VCG}_t^s(I \wedge b \wedge e_v = x_0, S_b, I \wedge e_v < x_0)$. □

[2] Auxiliary variables are used at the logical level only, and not as program variables.

$$\frac{}{\textbf{skip} \preceq C_1 \,;\, \ldots \,;\, C_n}$$

$$\frac{}{C_1 \,;\, \ldots \,;\, C_{i-1} \,;\, C_{j+1} \,;\, \ldots \,;\, C_n \preceq C_1 \,;\, \ldots \,;\, C_i \,;\, \ldots \,;\, C_j \,;\, \ldots \,;\, C_n} \quad (1 < i \leq j \leq n \text{ or } 1 \leq i \leq j < n)$$

$$\frac{C'_i \preceq C_i}{C_1 \,;\, \ldots \,;\, C'_i \,;\, \ldots \,;\, C_n \preceq C_1 \,;\, \ldots \,;\, C_i \,;\, \ldots \,;\, C_n} \quad (i \leq i \leq n)$$

$$\frac{S'_1 \preceq S_1 \qquad S'_2 \preceq S_2}{\textbf{if } b \textbf{ then } S'_1 \textbf{ else } S'_2 \preceq \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2}$$

$$\frac{S' \preceq S}{\textbf{while } b \textbf{ do } \{I\}\, S' \preceq \textbf{while } b \textbf{ do } \{I\}\, S}$$

Fig. 2. Definition of relation "is portion of"

Notation and Auxiliary Definitions. Let $S = C_1 \,;\, \ldots \,;\, C_n$, $1 \leq k \leq n$. We will use dedicated notation for the weak precondition of a suffix of S and the strong postcondition of a prefix of S, as well as for the (partial correctness) verification conditions of both, as follows.

$$\overline{\mathsf{wprec}}_k(S, Q) = \mathsf{wprec}(C_k \,;\, C_{k+1} \,;\, \ldots \,;\, C_n, Q) \qquad \overline{\mathsf{VC}}^w[k](S, Q) = \mathsf{VC}^w(C_k \,;\, C_{k+1} \,;\, \ldots \,;\, C_n, Q)$$
$$\overline{\mathsf{wprec}}_{n+1}(S, Q) = Q \qquad \overline{\mathsf{VC}}^w[n+1](S, Q) = \{\}$$

$$\overline{\mathsf{spost}}_0(S, P) = P \qquad \overline{\mathsf{VC}}^s[0](S, P) = \{\}$$
$$\overline{\mathsf{spost}}_k(S, P) = \mathsf{spost}(C_1 \,;\, \ldots \,;\, C_{k-1} \,;\, C_k, P) \qquad \overline{\mathsf{VC}}^s[k](S, P) = \mathsf{VC}^s(C_1 \,;\, \ldots \,;\, C_{k-1} \,;\, C_k, P)$$

$\overline{\mathsf{VC}}^w_t$ and $\overline{\mathsf{VC}}^s_t$ will also be used with the obvious meaning. We will additionally employ the following notation for the sequence obtained by removing a subsequence of S. For $1 \leq i \leq j \leq n$,

$$\mathsf{remove}(i, j, S) = \begin{cases} \textbf{skip} & \text{if } i = 1 \text{ and } j = n, \\ C_1 \,;\, \ldots \,;\, C_{i-1} \,;\, C_{j+1} \,;\, \ldots \,;\, C_n & \text{otherwise.} \end{cases}$$

Finally, we will write $S' \preceq S$ with the meaning that program S' results from S by removing some statements. S' is said to be a *portion* or a *reduction* of S.

Definition 1 (Portion-of relation). The $\cdot \preceq \cdot$ relation is the reflexive transitive closure of the relation generated by the set of axioms and rules given in Figure 2.

Note that since Figure 2 defines an anti-symmetric relation, $\cdot \preceq \cdot$ is a partial-order. As will be shortly seen, slices of a program S are portions of S that satisfy additional constraints.

```
1 x := x+100;
2 x := x+50;
3 x := x−100
```

Program 1: Example for postcondition-based slicing

3. Assertion-based Slicing: A Review

In this section we discuss the existent notions of slicing based on preconditions and postconditions, as well as algorithms for calculating them. Other related approaches are discussed in Section 3.5, in particular the notions of forward and backward *conditioned slice*.

We use the expression *assertion-based slicing* to encompass *postcondition-based*, *precondition-based*, and *specification-based* forms of slicing, which will be considered in turn in what follows. It is important to keep in mind the distinction between the definition of some form of slicing (which states when a program is a slice of another based on a given criterion), and algorithms for computing such slices. The fact that definitions and algorithms have often been introduced simultaneously in the same papers may cause some confusion between the two. Typically a definition admits more than one slice based on the same criterion, and an algorithm computes one particular slice in accordance with the definition.

This section is intended to introduce the reader to the key concepts of slicing based on assertions, but also to identify some limitations in the published work, which we then go on to solve in the rest of the paper.

3.1. Postcondition-based Slicing

The idea of slicing programs based on their specifications was introduced by Comuzzi et al. [CH96] with the notion of *predicate slice* (*p-slice*), also known as postcondition-based slice. To understand the idea of p-slices, consider a program S and a given postcondition Q. It may well be the case that some of the commands in the program do not contribute to the truth of Q in the final state of the program, i.e. their presence is not required in order for the postcondition to hold. In this case, the commands may be removed. A crucial point here is that the considered set of executions of the program is restricted to those that will result in the postcondition being satisfied upon termination. In other words, not every initial state is admissible – only those for which the *weak precondition* of the program with respect to Q holds.

Consider for instance Program 1. The postcondition $Q = x \geq 0$ yields the weak precondition $x \geq -50$. If the program is executed in a state in which this precondition holds and the commands in lines 2 and 3 are removed from it, the postcondition Q will still hold. To convince ourselves of this, it suffices to notice that after execution of the instruction in line 1 in a state in which the weak precondition is true, the condition $x \geq 50$ will hold, which is in fact stronger than Q.

To be more systematic, for a program of the form $C_1 ; \ldots ; C_n$ with postcondition Q, if $\models \overline{\mathsf{wprec}}_i(S, Q) \rightarrow \overline{\mathsf{wprec}}_j(S, Q)$, with $i < j$, the sequence $C_i ; \ldots ; C_{j-1}$ can be removed. In particular, if $\models \overline{\mathsf{wprec}}_i(S, Q) \rightarrow Q$, the sequence $C_i ; \ldots ; C_n$ can be removed. For the previous example we have

$$\overline{\mathsf{wprec}}_3(S, Q) = x \geq 100,$$
$$\overline{\mathsf{wprec}}_2(S, Q) = x \geq 50,$$
$$\overline{\mathsf{wprec}}_1(S, Q) = x \geq -50.$$

Now observe that $\models \overline{\mathsf{wprec}}_2(S, Q) \rightarrow Q$, which means that the instructions in lines 2 to 3 can in fact be removed: the postcondition Q will still hold for the sliced program when it is executed in a state satisfying $x \geq -50$.

P-slices are of course *not unique*. For instance since $\models \overline{\mathsf{wprec}}_3(S, Q) \rightarrow Q$ as well, we could have chosen to remove only the instruction in line 3. Informally we can say that given a set of slices of a program with respect to the same postcondition, the best slice is the one in which the largest number of instructions is removed. It is also important to understand that not only suffixes of a sequence of commands may be removed. Consider

```
1 x := x−150;
2 x := x+100;
3 x := x+100
```

Program 2: Example for postcondition-based slicing

the postcondition $Q = x \geq 0$ for Program 2, which yields the following weak preconditions

$\overline{\mathsf{wprec}}_3(S, Q) = x \geq -100$,
$\overline{\mathsf{wprec}}_2(S, Q) = x \geq -200$,
$\overline{\mathsf{wprec}}_1(S, Q) = x \geq -50$

Note that although $\not\models \overline{\mathsf{wprec}}_1(S, Q) \rightarrow Q$, the commands in lines 1 and 2 can be removed because $\models \overline{\mathsf{wprec}}_1(S, Q) \rightarrow \overline{\mathsf{wprec}}_3(S, Q)$. If the statement in line 3 is executed in a state in which $x \geq -50$ then the postcondition $x \geq 0$ will hold.

We remark that in the limit, the set of executions that lead to the postcondition being satisfied may be empty (if the weak precondition of the program is a contradiction, say $x < 0 \wedge x > 10$), in which case there exist no slices – the formal definition to be given below will clarify this point. Another extreme situation occurs when the postcondition is a valid assertion, say $x < 0 \vee x > -10$, in which case the entire program is seen as irrelevant, and admits as a slice the trivial program **skip**.

Calculating p-slices

It is easy to see how p-slices of a sequence of commands $S = C_1 ; \ldots ; C_n$ can be computed with respect to a postcondition Q. The first step is of course to calculate the weak preconditions $\overline{\mathsf{wprec}}_i(S, Q)$, for $1 \leq i \leq n$, and to store this information, say in the abstract syntax tree of S.

The next step is to iterate the following basic procedure that attempts to remove the subsequence $C_i ; \ldots ; C_{j-1}$, with $1 \leq i < j \leq n$:

- If $\models \overline{\mathsf{wprec}}_i(S, Q) \rightarrow \overline{\mathsf{wprec}}_j(S, Q)$ then slice S to $\mathsf{remove}(i, j-1, S)$

This involves a trade-off between the number of proof obligations generated (each of which results in a call to the prover) and the potential number of lines that will be removed from the program. Suppose for instance that we limit ourselves to removing suffixes of the initial program. The smallest such slice can be calculated with a linear number of calls to the prover (on the length of S), by fixing $j = n + 1$ (thus $\overline{\mathsf{wprec}}_j(S, Q) = Q$). It suffices, in the second step above, to initialize $i = 1$, and then execute the following loop: the prover is invoked with the formula $\overline{\mathsf{wprec}}_i(S, Q) \rightarrow Q$; if unsuccessful then i is incremented and a new iteration of the loop takes place; otherwise the algorithm stops. The resulting slice is $C_1 ; \ldots ; C_{i-1}$.

Notice that this is of course a conservative approach: failure of the prover to establish the validity of the first-order formula $\overline{\mathsf{wprec}}_i(S, Q) \rightarrow Q$ does not mean that the formula is not valid, but this uncertainty implies that removing the sequence $C_i ; \ldots ; C_n$ might result in a program that is not a slice of S, so the algorithm proceeds to the next candidate suffix.

It is easy to understand that the same program may contain several removable subsequences, including prefixes, suffixes, and sequences that are neither prefixes nor suffixes. Moreover, these removable sequences may well overlap. Thus it is clear that no linear-time algorithm can possibly detect all removable sequences, let alone select the smallest slice.

The Original Quadratic Time Algorithm

The algorithm proposed by Comuzzi runs in *quadratic time* on the length of the sequence.[3] The algorithm first tries to slice the entire program by removing its longest removable suffix, and then repeats this task,

[3] We remark that when an algorithm is said to run in quadratic time, we are referring to a count of the proof obligations generated to check whether a particular sequence of statements can be eliminated. In order for this to be reflected in an algorithm that actually runs in quadratic time, it is necessary to place a *time-out* limit for the external automated proof tool; this allows us to consider that proof obligations are discharged in constant time. We leave concrete provers and their usage out of the discussion.

```
1 x := x+100;
2 x := x−200;
3 x := x+200
```

Program 3: Example for precondition-based slicing

considering successively shorter prefixes of the resulting program, and removing their longest removable suffixes. Schematically:

for $j = n+1, n, \ldots, 2$
  for $i = 1, \ldots, j-1$
    if valid $\left(\overline{\mathsf{wprec}}_i(S,Q) \rightarrow \overline{\mathsf{wprec}}_j(S,Q)\right)$ **then** $S \leftarrow \mathsf{remove}(i, j-1, S)$

For instance in a program with 999 statements the following pairs (i, j) would be considered in this order:

$$(1,1000),(2,1000),\ldots,(999,1000),(1,999),(2,999),\ldots,(998,999),(1,998),\ldots$$

This algorithm may fail to remove the longest sequence. Consider that $\models \overline{\mathsf{wprec}}_1(S,Q) \rightarrow \overline{\mathsf{wprec}}_{800}(S,Q)$ and $\models \overline{\mathsf{wprec}}_{700}(S,Q) \rightarrow \overline{\mathsf{wprec}}_{900}(S,Q)$. Two subsequences may be sliced off, consisting respectively of commands 1 to 799 and 700 to 899. The algorithm will consider (and remove) the shorter sequence first, and in doing so will eliminate the possibility of the longer sequence being considered, since line 800 will be removed (and it may happen that $\overline{\mathsf{wprec}}_1(S,Q)$ is not stronger than any remaining $\overline{\mathsf{wprec}}_k(S,Q)$). The resulting slice is thus *not minimal*.

An Improved Quadratic Algorithm

An alternative to Comuzzi's algorithm can be described as follows. We start with the entire program and consider in turn successively shorter sequences as candidates to be removed. Thus in the 999 statements program one would consider sequences in the order $(1,1000),(1,999),(2,1000),(1,998),(2,999),(3,1000),(1,997),\ldots$ This would certainly remove the longest removable sequence.

This algorithm is however not optimal either. Consider the case in which $\models \overline{\mathsf{wprec}}_1(S,Q) \rightarrow \overline{\mathsf{wprec}}_{400}(S,Q)$, $\models \overline{\mathsf{wprec}}_{600}(S,Q) \rightarrow \overline{\mathsf{wprec}}_{1000}(S,Q)$, and $\models \overline{\mathsf{wprec}}_{200}(S,Q) \rightarrow \overline{\mathsf{wprec}}_{800}(S,Q)$. The longest sequence will be sliced off (600 program lines), but this will preclude the possibility of eliminating two shorter sequences that would together consist of 800 program lines: removing the larger contiguous sequence does not necessarily result in the smallest slice. In fact it should now be clear that considering all sequences in any given order cannot guarantee that the minimal slice is computed. The same is true for precondition-based and specification-based slices, discussed below. In Section 7 we will show that this problem can in general be formulated as a graph problem, which is one of the contributions of the present paper.

3.2. Precondition-based Slicing

Chung and colleagues [CLYK01] later introduced *precondition-based slicing* as the dual notion of postcondition-based slicing. The idea is still to remove statements whose presence does not affect properties of the final state of a program. The difference is that the considered set of executions of the program is now restricted directly through a first-order condition on the initial state. Statements whose absence does not violate any property of the final state of any such execution can be removed. This is the same as saying that the assertion calculated as the strong postcondition of the program (resulting from propagating forward the given precondition) is not weakened in the computed slice.

As an example of a precondition-based slice, consider now Program 3, and the precondition $P = x \geq 0$. The effect of the first two instructions is to weaken the precondition. If these instructions are sliced off and the resulting program is executed in a state in which P holds, whatever postcondition held for the initial program will still hold for the sliced program.

To be systematic, for a program of the form $C_1\,;\ldots;\,C_n$ with precondition P, if $\models \overline{\mathsf{spost}}_i(S,P) \rightarrow \overline{\mathsf{spost}}_j(S,P)$, with $i < j$, the sequence $C_{i+1}\,;\ldots;\,C_j$ can be removed. In particular, if $\models P \rightarrow \overline{\mathsf{spost}}_j(S,P)$,

```
1  if (x >= 0) then
2      x := x+100;
3      x := x−200;
4      x := x+200
5  else
6      x := x−150;
7      x := x−100;
8      x := x+100
9
10 ——————————————
11
12 if (x >= 0) then
13     x := x+200
14 else
15     skip
```

Program 4: Example for precondition-based slicing

the sequence $C_1 ; \ldots ; C_j$ can be removed. For the previous example we have

$$\begin{aligned}
\overline{\mathsf{spost}}_1(S,P) &= \exists v.v \geq 0 \wedge x = v + 100 && \equiv x \geq 100,\\
\overline{\mathsf{spost}}_2(S,P) &= \exists v.v \geq 100 \wedge x = v - 200 && \equiv x \geq -100,\\
\overline{\mathsf{spost}}_3(S,P) &= \exists v.v \geq -100 \wedge x = v + 200 && \equiv x \geq 100
\end{aligned}$$

We see that $\models P \rightarrow \overline{\mathsf{spost}}_2(S,P)$, thus the first two commands can be sliced off. Similarly to postcondition-based slicing, we are not limited to removing prefixes (even though only prefixes are considered by the linear time algorithm proposed in [CLYK01]). In the same example program, since in fact $\models \overline{\mathsf{spost}}_1(S,P) \rightarrow \overline{\mathsf{spost}}_3(S,P)$, we could alternatively slice off lines 2 and 3 of the program, which shows that removable sequences may overlap.

As a final example, consider a program containing branching, Program 4 (top). Again slicing the program involves computing its strong postcondition with respect to a given precondition P. Both branches consist of sequences of commands; even if the conditional command itself cannot be sliced off, it may well be the case that the branch subprograms can be sliced. To this effect, we strengthen the precondition with the boolean condition and its negation respectively, and slice each branch with respect to these strengthened preconditions. Let S_1 be x :=x+100; x := x−200; x := x+200 and S_2 be x := x−150; x := x−100; x := x+100. S_1 will be sliced with respect to $P_1 = P \wedge x \geq 0$ and S_2 with respect to $P_2 = P \wedge x \ngeq 0$.

Now let P be $x \geq 0$. Then $P_1 \equiv x \geq 0$ and P_2 is a contradiction, which means that $\models P_2 \rightarrow \mathsf{spost}(S_2, P_2)$. Consequently, S_2 will be sliced to **skip**. This makes sense, since the precondition eliminates the possibility of execution of the *else* branch of the conditional. On the other hand the *then* branch is just the previous example (Program 3). Thus Program 4 can be precondition-sliced with respect to $x \geq 0$ as shown at the bottom.

3.3. Specification-based Slicing

A *specification-based slice* can be calculated when both a precondition P and a postcondition Q are given for a program S. The set of relevant executions is restricted to those for which Q holds upon termination when the program is executed in a state satisfying P. Programs resulting from S by removing a set of statements, and which are still correct regarding (P, Q), are said to be specification-based slices of S with respect to (P, Q).

The method proposed in [CLYK01] to compute such slices is based on a theorem proved by the authors, which states that the composition, in any order, of postcondition-based slicing (with respect to postcondition Q) and precondition-based slicing (with respect to precondition P) produces a specification-based slice with respect to (P, Q). As an example consider Program 5 and the specification $(y > 10, x \geq 0)$. Precondition-based slicing will slice both sequences inside the conditional by strengthening the precondition $y > 10$ with

```
1  if (y > 0) then
2      x := 100;
3      x := x+50;
4      x := x−100
5  else
6      x := x−150;
7      x := x−100;
8      x := x+100
9
10 ——————————————
11
12 if (y > 0) then
13     x := 100
14 else
15     skip
```

Program 5: Example for specification-based slicing

```
1 x := x∗x;
2 x := x+100;
3 x := x+50
```

Program 6: Example for specification-based slicing

the condition $y > 0$ and its negation respectively. In the second case this yields a contradiction, which will result in the *else* branch sequence being completely sliced off. The *then* sequence branch is not affected. Postcondition-based slicing with respect to $x \geq 0$ will then produce the sliced program shown at the bottom of the listing.[4]

Although this method does compute specification-based slices, it does not compute minimal slices, as can be seen by looking at Program 6 with specification $(\mathsf{true}, x \geq 100)$. We have:

$$\overline{\mathsf{spost}}_0(S,P) = \mathsf{true}$$
$$\overline{\mathsf{spost}}_1(S,P) = \exists v.x = v * v$$
$$\overline{\mathsf{spost}}_2(S,P) = \exists w.(\exists v.w = v * v) \wedge x = w + 100 \qquad \equiv \exists v.x = v * v + 100$$
$$\overline{\mathsf{spost}}_3(S,P) = \exists w.(\exists v.w = v * v + 100) \wedge x = w + 50 \qquad \equiv \exists v.x = v * v + 150$$

and

$$\overline{\mathsf{wprec}}_4(S,Q) = x \geq 100 = Q$$
$$\overline{\mathsf{wprec}}_3(S,Q) = x \geq 50$$
$$\overline{\mathsf{wprec}}_2(S,Q) = x \geq -50$$
$$\overline{\mathsf{wprec}}_1(S,Q) = \mathsf{true}$$

It is obvious that the postcondition is satisfied after execution of the instruction in line 2, which means that if line 3 is removed the sliced program will still be correct with respect to $(\mathsf{true}, x \geq 100)$. However, precondition-based and postcondition-based slicing both fail in removing this instruction, since no forward implications are valid among the $\overline{\mathsf{spost}}_i(S,P)$ or the $\overline{\mathsf{wprec}}_i(S,Q)$. Composing precondition-based and postcondition-based slicing will of course not solve this fundamental flaw. In Section 5 we show that the precise identification of removable statements requires the *simultaneous* use of both preconditions and postconditions; trying to identify removable statements using only preconditions or only postconditions may fail.

[4] in fact [CLYK01] advocates replacing the entire conditional command by one of the branches when the other branch is sliced to **skip**, but it is debatable whether this transformation can still be considered as a form of slicing.

3.4. Formalization

We now formalize the notions of slicing reviewed in this section. A program S' is a *specification-based slice* of S if it is a *portion* of S and moreover S can be *refined* to S' with respect to a given specification (a semantic notion). The notions of precondition-based and postcondition-based slice can be defined as special cases of this notion.

Definition 2 (Assertion-based slices). Let S be a program and (P, Q) a specification consisting of precondition P and postcondition Q. The program S' is said to be

- a *specification-based slice* of S with respect to (P, Q), written $S' \lhd_{(P,Q)} S$, if $S' \preceq S$ and
$$\models \mathsf{VCG}^w(P,\, S,\, Q) \quad \text{implies} \quad \models \mathsf{VCG}^w(P,\, S',\, Q)$$
- a *precondition-based slice* of S with respect to P, if $S' \lhd_{(P,\mathsf{spost}(S,P))} S$;
- a *postcondition-based slice* of S with respect to postcondition Q if $S' \lhd_{(\mathsf{wprec}(S,Q),Q)} S$.

Observe that it only makes sense to calculate specification-based slices of correct programs; if S is not correct with respect to (P, Q) then any portion of it is a slice with respect to (P, Q). This does not however mean that techniques based on these forms of slicing cannot be applied to incorrect programs: they can be used on subprograms (proved correct) of incorrect programs. For instance in Section 4 we will see how postcondition-based slicing can be used for debugging purposes.

Note also that the definitions of precondition-based and postcondition-based slicing are very strong, as the following lemma shows.

Lemma 3.

1. If S' is a *precondition-based slice* of S with respect to P, then for any assertion Q, $S' \lhd_{(P,Q)} S$
2. If S' is a *postcondition-based slice* of S with respect to Q, then for any assertion P, $S' \lhd_{(P,Q)} S$

Proof. We prove 1 (the proof of 2 is similar). We assume $\models \mathsf{VCG}^w(P,\, S,\, Q)$, and thus by Lemma 1 $\models \mathsf{spost}(S, P) \rightarrow Q,\ \mathsf{VC}^s(S, P)$, and have to prove that $\models \mathsf{VCG}^w(P,\, S',\, Q)$. Since S' is a precondition-based slice of S with respect to P, by Lemma 1 we have that

$$\models \mathsf{spost}(S, P) \rightarrow \mathsf{spost}(S, P),\ \mathsf{VC}^s(S, P) \text{ implies } \models \mathsf{spost}(S', P) \rightarrow \mathsf{spost}(S, P),\ \mathsf{VC}^s(S', P)$$

The left-hand side follows from our assumptions, and thus

$$\models \mathsf{spost}(S', P) \rightarrow Q,\ \mathsf{VC}^s(S', P)$$

which by the same lemma is equivalent to $\models \mathsf{VCG}^w(P,\, S',\, Q)$. □

We must remark at this point that there are several differences between our definitions and those used in [CH96, CLYK01]. A first difference concerns all the above notions of slicing: previous notions require the weak precondition (resp. strong postcondition) to be *exactly the same* in the sliced program as in the original program, whereas we allow for it to be *weaker* (resp. *stronger*), which is more coherent with the idea of the slice refining the behaviour of the original program.

A second difference concerns specifically the definitions of precondition-based and postcondition-based slicing only. While the definitions given in [CLYK01] are based on implicative assertions relating the strong postconditions (resp. weak preconditions) of both programs, we explicitly define them as particular cases of specification-based slices, which is more convenient given our treatment of iteration through the use of annotated invariants. The following lemma makes the relation between both definitions explicit, for the case of programs without iteration.

Lemma 4. Let S' be a program containing no loops. Then

1. If $S' \preceq S$ and $\models \mathsf{spost}(S', P) \rightarrow \mathsf{spost}(S, P)$, then $S' \lhd_{(P,\mathsf{spost}(S,P))} S$.
2. if $S' \preceq S$ and $\models \mathsf{wprec}(S, Q) \rightarrow \mathsf{wprec}(S', Q)$, then $S' \lhd_{(\mathsf{wprec}(S,Q),Q)} S$

Proof.

1. Note that $\mathsf{VCG}^s(P,\, S,\, \mathsf{spost}(S, P)) = \mathsf{spost}(S, P) \rightarrow \mathsf{spost}(S, P)$, which is valid, and $\mathsf{VCG}^s(P,\, S',\, \mathsf{spost}(S, P)) = \mathsf{spost}(S', P) \rightarrow \mathsf{spost}(S, P)$. Thus $\models \mathsf{VCG}^s(P,\, S,\, \mathsf{spost}(S, P))$ implies $\models \mathsf{VCG}^s(P,\, S',\, \mathsf{spost}(S, P))$ and by Lemma 1 we have that $\models \mathsf{VCG}^w(P,\, S,\, \mathsf{spost}(S, P))$ implies $\models \mathsf{VCG}^w(P,\, S',\, \mathsf{spost}(S, P))$

2. Similar to 1.

□

The above definitions are formulated in a partial correctness setting, which means that terminating programs admit non-terminating programs as specification-based slices, and vice versa (it is easy to see that removing a single instruction from the body of a terminating loop may make it non-terminating, and vice versa). We will now introduce *termination-sensitive* notions of slicing, by shifting from a partial correctness to a total correctness setting. A terminating program does not admit non-terminating programs as termination-sensitive slices.

Definition 3 (Termination-sensitive assertion-based slices). Let S be a program and (P, Q) a specification consisting of precondition P and postcondition Q. The program S' is said to be

- a *termination-sensitive specification-based slice* of S with respect to (P, Q), written $S' \blacktriangleleft_{(P,Q)} S$, if $S' \preceq S$ and moreover
$$\models \mathsf{VCG}_t^w(P, S, Q) \quad \text{implies} \quad \models \mathsf{VCG}_t^w(P, S', Q)$$
- a *termination-sensitive precondition-based slice* of S with respect to P if $S' \blacktriangleleft_{(P,\mathsf{spost}(S,P))} S$;
- a *termination-sensitive postcondition-based slice* of S with respect to Q if $S' \blacktriangleleft_{(\mathsf{wprec}(S,Q),Q)} S$.

In the same way that, when verifying a program, one may proceed by first checking its partial correctness and then its termination to ensure total correctness, one may assert that S' is a termination-sensitive slice of the totally correct program S by checking that $S' \triangleleft_{(P,Q)} S$ and additionally checking that S' terminates.

3.5. Related Approaches

In this section we review other forms of slicing that have some points in common with assertion-based slicing.

Semantic Slicing

One of the most successful lines of work in the area of slicing has been conducted by Ward and colleagues. This line has focused on semantic forms of slicing, in the sense that slices are obtained by combining syntactic operations with classic semantics-preserving *program transformations* such as loop unrolling and constant propagation. The results are both practical (a commercially-available workbench has been developed) and theoretical. In particular, the recent paper [War09] provides a clarifying analysis of slicing properties and definitions proposed by different authors (both syntactic and semantic). Our work in this paper clearly stands on the semantic side, but a fundamental difference with respect to other work on semantic slicing is that we focus on code annotated with assertions. Our slicing criteria are exclusively provided by such assertions.

Conditioned Slicing

Shortly after the definition of postcondition-based slicing by Comuzzi and Hart, Canfora et al. [CCL98] introduced the notion of *conditioned slicing*, together with a tool to calculate such slices. Similarly to precondition-based slicing, conditioned slicing uses preconditions as a means to specify a set of initial states for computing a forward slice. The main points to understand about conditioned slicing are

1. The precondition is used in combination with traditional slicing techniques based on dependency analysis. Code will be removed either because it is unreachable (not executed when the program is started in a state in which the precondition holds), or because it is dead (the precondition eliminates dependencies involving it). Consider the following example from Canfora's paper:

```
1 x := y+2;
2 if (a > 0)
3   x := y*2;
4 z := x+1
```

A conditioned slice of this program based on any precondition P such that $\models P \rightarrow a > 0$ results in line

1 being eliminated, since line 3 will certainly be executed and cancel the effect of line 1. This example shows a fundamental difference between conditioned slicing and earlier notions of slicing exclusively based on control and data dependencies. Clearly static dependencies alone cannot be used to implement conditioned slicing, since the instruction in line 4 depends on all previous instructions. The algorithm proposed by the authors is based on *symbolic execution*, which allows for the relevant dependency paths to be identified. A theorem prover is called externally to guide the symbolic execution.

2. In the context of traditional, dependency-based slicing, there are two standard types of forward slicing: *static*, which considers every possible execution (i.e. all initial states), and *dynamic*, which is concerned with a single execution (a concrete initial state) of the program. The latter can be generalized to cope with a set of concrete executions, but an interesting aspect of conditioned slicing is that it subsumes all these notions, since a characterization of the set of initial states by a first-order condition can be used to admit any initial state (if the condition is `true`), or just a concrete initial state (if the condition is a conjunction of equality formulas, each equating a program variable to a constant), or any other intermediate set of initial states.
3. The similarities between precondition-based slicing and conditioned slicing should be clear: even though the latter is based on dependencies and the former on weak preconditions and strong postconditions, both are capable of eliminating conditionally unreachable and conditionally dead code. These are examples of code that is redundant with respect to a given precondition, but note that the notion of redundancy is different in both cases: whereas in precondition-based slicing this is code that, if removed, results in a program whose strong postcondition will not be weakened with respect to the initial program, in conditioned slicing this is code that does not contribute to the values of a given set of variables. Precondition-based slicing removes other forms of redundancy that conditioned slicing cannot remove, since they can only be detected at a semantic level. Program 4 is a good example to illustrate this point: while conditioned slicing with $x \geq 0$ would eliminate the *else* branch, it would not remove the two assignment commands inside the *then* branch, which are removed by precondition-based slicing.
4. Conditioned slicing criteria are not however limited to a precondition P: a slicing criterion consists additionally of a subset X of the program variables, as well as a specific program line k. The program statements eliminated are those that do not affect the value of any variable in X at line k, for executions starting in states satisfying P. Precondition-based slicing does not subsume conditioned slicing, since it does not take into account these criteria, inherited from standard dependency-based forms of slicing (see also Section 8 below).
5. For conditioned slicing criteria that focus on the final state of the program (i.e. k is the last line), precondition-based slicing can be said to be a stronger form of slicing than conditioned slicing, since it eliminates code using semantic criteria that cannot be expressed in terms of dependencies.

Backward Conditioned Slicing

Backward conditioning was introduced by Fox and colleagues [FDHH01] as the symmetric notion of conditioned slicing. A slicing criterion includes a postcond Q that is used in the following way: statements whose presence *forces* $\neg Q$ to hold in the final state (i.e. if they are present $\neg Q$ will hold after every execution) are removed.

The technique is intended as the dual of conditioned slicing: whereas (forward) conditioned slicing eliminates the code that will surely not be executed when the given precondition holds, backward conditioned slicing eliminates the code that cannot be executed if the given postcondition is to be satisfied in the final state, i.e. it eliminates statements that prevent the given postcondition from being true. The technique is introduced with program comprehension as main application. The authors also propose an algorithm for implementing backward conditioned slicing, based on symbolic execution and an external theorem prover.

A second paper [HHF$^+$01] combines forward and backward conditioned slicing, based on a precondition P and a postcondition Q: it eliminates code that leads to Q being false when the program is executed in states satisfying P. The motivation of the latter work is the application to program verification. The idea here is that to check if a program is correct w.r.t. a specification (P, Q), one may compute its conditioned slice w.r.t. $(P, \neg Q)$. If the program is correct this slice will be *empty*, since all execution paths lead to Q being true, and all instructions will thus be removed. If the program is not correct, the instructions that remain in the slice are those that *may for some initial states* lead to $\neg Q$ being true. Such instructions should carefully be considered since they are directly contributing to the program being incorrect.

This forward/backward form of conditioned slicing cannot be formulated as specification-based slicing with respect to a specification. While a specification-based slice S' of program S with respect to (P, Q) is correct with respect to (P, Q), a conditioned slice S'' with respect to (P, Q) is characterized by *not being correct* with respect to $(P, \neg Q)$. Another way to put this is that while we require $\mathsf{VCG}^w(P,\ S',\ Q)$ to be *valid*, $\mathsf{VCG}^w(P,\ S'',\ Q)$ must instead be *satisfiable*. For instance the command $x := x + 10$ with precondition $x > 10$ and postcondition $x \leq 20$ should be sliced to **skip**, since $\models \mathsf{VCG}^w(x > 10,\ x := x + 10,\ \neg(x \leq 20))$ and $\not\models \mathsf{VCG}^w(x > 10,\ \mathbf{skip},\ \neg(x \leq 20))$, i.e. the formulas $\mathsf{VCG}^w(x > 10,\ \mathbf{skip},\ x \leq 20)$ are satisfiable.

We speculate that the graph-based algorithms studied in this paper could be adapted to the purpose of computing forward/backward conditioned slices by using satisfiability checks instead of validity checks.

In the next section, before focusing on the core contributions of the paper, we will briefly consider some applications of slicing based on assertions.

4. Applications of Assertion-based Slices

Postcondition-based and precondition-based slicing can both be seen as program specialization techniques regarding a restricted set of executions of a program. A postcondition-based slice of a program S may have a *weaker weak precondition* than S, and a precondition-based slice of S may have a *stronger strong postcondition* than S. We start by considering applications of these forms of slicing and then turn to specification-based slicing.

Postcondition-based Slicing

In their paper Comuzzi and Hart give a number of examples of the usefulness of postcondition-based slicing, based on their experience as software developers and maintainers. Their emphasis is on applying slicing to relatively small fragments of big programs, using postconditions corresponding to properties that should be *preserved* by these fragments. Suppose one suspects that a problem was caused by some property Q being false at line k of a program S with n lines of code. We can take the subprogram S_k consisting of the first k lines of S and slice it with respect to the postcondition Q. This may result in a suffix of S_k being sliced off, say from lines i to k, which means that in order for Q to hold at line k, it must also hold at line i. The resulting slice is where the software engineers should now concentrate in order to find the problem (a similar reasoning applies if the sequence of lines removed is not a suffix of S_k).

A related situation occurs when the property must deliberately be violated in some part of the code. This is typical for instance of code running as a thread of a concurrent program, with Q being true outside a critical section executed by the thread at some point, and false inside that section. Q is true before entering the said critical section and will be true after leaving it, so postcondition-based slicing can be used to study the correct behaviour of the code with respect to that section. Similarly, the property may correspond to some invariant of a data structure, say a balanced binary search tree that will temporarily be unbalanced (or even inconsistent) while a new element is being inserted.

Safety properties may also be studied in this way. Examples include for instance

- array accesses $u[e]$, with safety property "the value of expression e stands between 0 and $N - 1$", with N the allocated size of the array;
- pointer dereferencing accesses $*p$ with safety property "p points to a properly allocated memory region";
- procedure invocations, with safety property "the precondition of the invoked procedure is satisfied".

Precondition-based Slicing

Redundant code is code that does not produce any effect: removing it results in a program that behaves in the same way as the original. Note that we say "the code does not produce any effect" in the sense of observable effects on the final state. Removing redundant code may of course result in code that is different regarding the execution traces; in particular the resulting code may be faster to execute. A major application of precondition-based slicing is the removal of *conditionally redundant code*, i.e. code that is redundant for executions of the program specified by a given precondition. Naturally, redundant code is a special case of conditionally redundant code.

```
1 x := y + 2;
2 x := y * 2;
```

Program 7: Example for dead code elimination by precondition-based slicing

Examples of redundancies include sequences of instructions like $x := x - 200\,; x := x + 200$, as in Program 3. Previously in Section 3 we saw how precondition-based slicing with respect to the precondition $x \geq 0$ indeed removed these two instructions. It is however clear that the instructions should be removable also for executions not allowed by this precondition. Let us now consider how this can be done.

A first attempt could be to slice the program with respect to the precondition true. For Program 3 we would have

$$\begin{aligned}
\overline{\mathsf{spost}}_1(S, \mathsf{true}) &= \exists v.x = v + 100 && \equiv \mathsf{true},\\
\overline{\mathsf{spost}}_2(S, \mathsf{true}) &= \exists v.x = v - 200 && \equiv \mathsf{true},\\
\overline{\mathsf{spost}}_3(S, \mathsf{true}) &= \exists v.x = v + 200 && \equiv \mathsf{true}
\end{aligned}$$

the entire program can now be sliced off, since its calculated postcondition is a valid assertion – not what we had in mind. What is missing here is a way to record the initial state, to be able to compare the values of variables in different states using the initial values as a reference. For this purpose we resort to *auxiliary variables*, that are used in assertions only, not in the code. The use of these variables makes postcondition calculations resemble a *symbolic execution* of the code, in which the values of the variables after the execution of each command are related to the initial values through equality formulas.

Let us slice the same program with respect to the precondition $x = x_0$, where the auxiliary variable x_0 is used to record the initial value of x:

$$\begin{aligned}
\overline{\mathsf{spost}}_1(S, x = x_0) &= \exists v.v = x_0 \wedge x = v + 100 && \equiv x = x_0 + 100,\\
\overline{\mathsf{spost}}_2(S, x = x_0) &= \exists v.v = x_0 + 100 \wedge x = v - 200 && \equiv x = x_0 - 100,\\
\overline{\mathsf{spost}}_3(S, x = x_0) &= \exists v.v = x_0 - 100 \wedge x = v + 200 && \equiv x = x_0 + 100
\end{aligned}$$

Notice that the precondition does not restrict the set of executions, since x_0 is not a program variable. Since $\models \overline{\mathsf{spost}}_1(S, \mathsf{true}) \rightarrow \overline{\mathsf{spost}}_3(S, \mathsf{true})$, the statements in lines 2 and 3 of the program can be sliced off, because they are redundant and unnecessary in *any* execution of the program.

Two particular forms of redundant code are *unreachable code*, which is not executed, and *dead code*, which is executed but produces no effect, because the final values of variables do not depend on its results. An example of unreachable code is the block S_2 in **if** $10 > 5$ **then** S_1 **else** S_2; an example of dead code is the first instruction in Program 7. Unreachable code and dead code elimination are typically part of the optimizations performed by compilers, using control flow and data flow analyses. Specification-base slicing allows for conditional versions of these notions, eliminating code that is unreachable or dead for a given set of executions. Conditional unreachable code elimination was already exemplified with Program 4 in Section 3.2 – unreachable code is eliminated because (for the given initial states) its presence does not influence the final state of the program. Precondition-based slicing can thus be used to study the control flow of a program.

Let us now consider an example of (unconditional) dead code elimination. So far we have been identifying slices by checking the validity of implicative formulas involving propagated strong postconditions. As hinted in the previous section, this technique cannot eliminate all types of redundant code, which this example will also illustrate. Let S be Program 7. Clearly, in every execution of this program, the first statement is dead since its effect is cancelled by the second statement. To slice this program using the precondition $x = x_0 \wedge y = y_0$ we compute the strong postconditions as follows:

$$\begin{aligned}
\overline{\mathsf{spost}}_0(S, x = x_0 \wedge y = y_0) &= x = x_0 \wedge y = y_0\\
\overline{\mathsf{spost}}_1(S, x = x_0 \wedge y = y_0) &= \exists v.v = x_0 \wedge y = y_0 \wedge x = y + 2 && \equiv x = y_0 + 2 \wedge y = y_0,\\
\overline{\mathsf{spost}}_2(S, x = x_0 \wedge y = y_0) &= \exists v.v = y_0 + 2 \wedge y = y_0 \wedge x = y * 2 && \equiv x = y_0 * 2 \wedge y = y_0
\end{aligned}$$

Since $\not\models \overline{\mathsf{spost}}_0(S, x = x_0 \wedge y = y_0) \rightarrow \overline{\mathsf{spost}}_1(S, x = x_0 \wedge y = y_0)$, the first statement cannot be sliced off. Clearly, it would be impossible to reach the conclusion that this statement is dead by relating the calculated

```
1 x := y+2;
2 if (a > 0)
3    then x := y*2;
4    else skip;
5 z := x+1
```

Program 8: Example for redundant code elimination by specification-based slicing

strong postconditions only, since $\overline{\mathsf{spost}}_0(S, x = x_0 \wedge y = y_0)$ and $\overline{\mathsf{spost}}_1(S, x = x_0 \wedge y = y_0)$ are calculated without even looking at subsequent commands. Below we will see that using a specification-based slicing technique to compute precondition-based slices will allow for commands like these to be removed as expected.

Specification-based Slicing

A first application of this form of slicing concerns the removal of redundant code. We saw above that the traditional precondition-based slicing algorithm is unable to remove all such code, specifically when a "look ahead" would be required to reach the conclusion that a given statement can be removed. Specification-based slicing combines strong postcondition with weak precondition computations, and can be used to properly eliminate redundant code. Let S be a program with variables $x^1, \ldots, x^n$. In order to remove unnecessary code taking into account every execution of S, it suffices to slice S with respect to the following specification (the x_0^i are auxiliary variables).

$$(x^1 = x_0^1, \ldots, x^n = x_0^n \ , \ \mathsf{spost}(S, x_1 = x_0^1, \ldots, x^n = x_0^n))$$

Note that, following Definition 2 (2), the result will still be a precondition-based slice – the problem of our previous attempt was not in the definition of precondition-based slice, but in the method used to compute these slices.

To illustrate this we consider again Program 7, and compute a precondition-based slice with respect to $x = x_0 \wedge y = y_0$, followed by a postcondition-based slice with respect to the strong postcondition $\mathsf{spost}(S, x = x_0 \wedge y = y_0)$. We saw before that the first step is unable to remove any statements in this example. The calculated postcondition is

$$Q = \mathsf{spost}(S, x = x_0 \wedge y = y_0) \equiv \ x = y_0 * 2 \wedge y = y_0$$

We now calculate weak preconditions using the above as postcondition:

$\overline{\mathsf{wprec}}_2(S, Q) = y * 2 = y_0 * 2 \wedge y = y_0$,
$\overline{\mathsf{wprec}}_1(S, Q) = y * 2 = y_0 * 2 \wedge y = y_0$

Now since $\models \overline{\mathsf{wprec}}_1(S, Q) \rightarrow \overline{\mathsf{wprec}}_2(S, Q)$, the statement in line 1 can indeed be removed, as would be expected.

This example motivates the following: henceforth in this paper we will concentrate on methods for calculating specification-based slices. Whenever the specification consists of a precondition (resp. postcondition) only, it will be completed by computing the strong postcondition (resp. weak precondition) of the program with respect to it. Computing precondition or postcondition-based slices as special cases of specification-based slices allows for a more precise identification of removable statements.

Program 8 is a further example of a precondition-based slice that will be calculated as a specification-based slice, following the ideas outlined above (it is taken from [CCL98], see Section 3.5). The idea is to slice this program with respect to the precondition $a > 0$. Clearly the *else* branch is dead, and if it was not already **skip** it would be replaced by **skip** in the computed slice, since it will not be executed with this precondition. The goal here is to illustrate something else: since the *then* branch will be executed, the first statement in the program will not produce any effect, since the final value of x will be given by the statement in line 3. It is thus a dead statement that should be eliminated.

Let S be the above program and P be $x = x_0 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0$. In order to eliminate redundant code we will calculate a slice of this program with respect to the specification $(P, \mathsf{spost}(S, P))$. We start by propagating the precondition forward using strong postcondition calculations, and then propagate

backward the strong postcondition, using weak precondition calculations. This is shown in Figure 3, in which we also simplify the calculated assertions to equivalent formulas. Clearly $\not\models \overline{\mathsf{spost}}_0(S,P) \rightarrow \overline{\mathsf{spost}}_1(S,P)$, but $\models \overline{\mathsf{wprec}}_1(S,P) \rightarrow \overline{\mathsf{wprec}}_2(S,P)$, thus the first command in the program can be eliminated.

Design by Contract and Specification-based Slicing. It may be useful to apply specification-based slicing to code already annotated with specifications, following the principles of *design by contract* – a software development approach that advocates specifying the behavior of program routines through the use of annotations, and checking them individually (either statically or dynamically) to obtain globally correct programs. For code that has been developed in this way, it is cheap to apply specification-based slicing techniques, based on the specification information that is already present in the code.

A first application in this context is again the elimination of unnecessary code. A piece of software that has already been proven correct with respect to a specification may well contain code that is not actually playing any useful role regarding that specification. This unnecessary code that may have been introduced during development is not detected by the verification process itself, but slicing the program with respect to the proven specification will hopefully remove such code.

A different application is concerned with code that has been verified but is now being used in a specialized context, i.e. the specification that is required for a given use of the code is actually weaker (because a stronger precondition is present, and/or a weaker postcondition is required) than the proven specification. A typical situation is software reuse. Think for instance of a library containing a procedure that implements a traversal of some data structure, and collects a substantial amount of information in that traversal. It may be the case that for a given project one wants to reuse this procedure without requiring all the information collected in the traversal. In this case the procedure will be invoked with a weaker specification, and it makes sense to produce a specialized version to be included in the current project. A specification-based slice can be computed to this effect.

5. Properties of Assertion-based Slicing

In abstract terms, given a program $S = C_1 ; \ldots ; C_n$ with specification (P, Q), an assertion-based slicing algorithm must be able to

1. Identify subprograms that *could* be removed from the program being sliced, while preserving its correctness with respect to a given specification. More concretely, the algorithm must decide for every i, j if $\mathsf{remove}(i, j, S) \triangleleft_{(P,Q)} S$ holds, and then proceed recursively to identify removable subprograms of each C_i.
2. Select, among the set of statements identified as removable, the combination (or one of the combinations) that results in the best slice according to some criterion (the most obvious is the smallest number of program lines). Although this has been considered more seriously in the work of Comuzzi and colleagues on postcondition-based slicing, it applies to all three forms of slicing we have considered.

This section is devoted to point 1; the second point will be considered in Section 7.

The currently available algorithms for precondition-based and postcondition-based slicing check the validity of a formula relating the propagated conditions near the statements i and j. This seemed to be a good test of whether the sequence of commands between i and j could be removed, but in Section 4 it was shown that for precondition-based slicing the method used in previous work fails to identify statements that should be removed because they do not contribute to the final state of the program, in any of the executions specified by the precondition. The failure occurs when the commands are made irrelevant by other instructions that occur *later* in the program, and thus cannot be detected by the prescribed method. The bottom line is that using Lemma 4 to design precondition or postcondition-based slicing algorithms is in fact misleading. The problem can be solved by simply observing our definition of these slices (Definition 2), which are given as particular cases of specification-based slices. For instance given a precondition P, it suffices to calculate the strong postcondition of the program with respect to P and then calculate a specification-based slice of S with respect to $(P, \mathsf{spost}(S, P))$.

For specification-based slicing, the algorithm of [CLYK01] considers sequentially the propagation of preconditions and postconditions. But in Section 3.3 we have already shown that first slicing with preconditions and later with postconditions (or vice versa) may fail to remove statements which can be removed, according

Forward propagation of precondition P:

$$
\begin{aligned}
&\overline{\mathsf{spost}}_0(S,P) = && x = x_0 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0\\
&\overline{\mathsf{spost}}_1(S,P) = && x = y_0 + 2 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0\\
&\mathsf{spost}(x := y * 2, a > 0 \wedge \overline{\mathsf{spost}}_1(S,P)) = && a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0\\
&\mathsf{spost}(\mathbf{skip}, \neg a > 0 \wedge \overline{\mathsf{spost}}_1(S,P)) = && \neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0\\
&\overline{\mathsf{spost}}_2(S,P) = && (a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0)\\
& && \vee\ (\neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0)\\
&\overline{\mathsf{spost}}_3(S,P) = && (a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0)\\
& && \vee\ (\neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \wedge z = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0)
\end{aligned}
$$

Backward propagation of postcondition $\overline{\mathsf{spost}}_3(S,P)$:

$$
\begin{aligned}
&\overline{\mathsf{wprec}}_4(S,P) = \overline{\mathsf{spost}}_3(S,P) = && (a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0)\\
& && \vee\ (\neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \wedge z = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0)\\
&\overline{\mathsf{wprec}}_3(S,P) = && (a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge x + 1 = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0)\\
& && \vee\ (\neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \wedge x + 1 = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0)\\
&\mathsf{wprec}(x := y * 2, \overline{\mathsf{wprec}}_3(S,P)) = && (a > 0 \wedge y * 2 = y_0 * 2 \wedge y = y_0 \wedge y * 2 + 1 = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0)\\
& && \vee\ (\neg a > 0 \wedge y * 2 = y_0 + 2 \wedge y = y_0 \wedge y * 2 + 1 = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0)\\
&\mathsf{wprec}(\mathbf{skip}, \overline{\mathsf{wprec}}_3(S,P)) = && (a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge x + 1 = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0)\\
& && \vee\ (\neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \wedge x + 1 = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0)\\
&\overline{\mathsf{wprec}}_2(S,P) = && (a > 0 \rightarrow (a > 0 \wedge y * 2 = y_0 * 2 \wedge y = y_0 \wedge y * 2 + 1 = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0)\\
& && \vee\ (\neg a > 0 \wedge y * 2 = y_0 + 2 \wedge y = y_0 \wedge y * 2 + 1 = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0))\\
& && \wedge\ (\neg a > 0 \rightarrow (a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge x + 1 = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0)\\
& && \vee\ (\neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \wedge x + 1 = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0))\\
&\overline{\mathsf{wprec}}_1(S,P) = && (a > 0 \rightarrow (a > 0 \wedge y * 2 = y_0 * 2 \wedge y = y_0 \wedge y * 2 + 1 = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0)\\
& && \vee\ (\neg a > 0 \wedge y * 2 = y_0 + 2 \wedge y = y_0 \wedge y * 2 + 1 = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0))\\
& && \wedge\ (\neg a > 0 \rightarrow (a > 0 \wedge y + 2 = y_0 * 2 \wedge y = y_0 \wedge y + 2 + 1 = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0)\\
& && \vee\ (\neg a > 0 \wedge y + 2 = y_0 + 2 \wedge y = y_0 \wedge y + 2 + 1 = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0))
\end{aligned}
$$

Simplified conditions:

$$
\begin{aligned}
&\overline{\mathsf{spost}}_0(S,P) \equiv && x = x_0 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0\\
&\overline{\mathsf{spost}}_1(S,P) \equiv && x = y_0 + 2 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0\\
&\mathsf{spost}(x := y * 2, a > 0 \wedge \overline{\mathsf{spost}}_1(S,P)) \equiv && a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0\\
&\mathsf{spost}(\mathbf{skip}, \neg a > 0 \wedge \overline{\mathsf{spost}}_1(S,P)) \equiv && \mathsf{false}\\
&\overline{\mathsf{spost}}_2(S,P) \equiv && a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0\\
&\overline{\mathsf{spost}}_3(S,P) \equiv && a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = y_0 * 2 + 1 \wedge a = a_0
\end{aligned}
$$

$$
\begin{aligned}
&\overline{\mathsf{wprec}}_4(S,P) = \overline{\mathsf{spost}}_3(S,P) \equiv && a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = y_0 * 2 + 1 \wedge a = a_0\\
&\overline{\mathsf{wprec}}_3(S,P) \equiv && a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge x + 1 = y_0 * 2 + 1 \wedge a = a_0\\
&\mathsf{wprec}(x := y * 2, \overline{\mathsf{wprec}}_3(S,P)) \equiv && a > 0 \wedge y * 2 = y_0 * 2 \wedge y = y_0 \wedge y * 2 + 1 = y_0 * 2 + 1 \wedge a = a_0\\
&\mathsf{wprec}(\mathbf{skip}, \overline{\mathsf{wprec}}_3(S,P)) \equiv && a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge x + 1 = y_0 * 2 + 1 \wedge a = a_0\\
&\overline{\mathsf{wprec}}_2(S,P) = && = a > 0 \wedge y * 2 = y_0 * 2 \wedge y = y_0 \wedge y * 2 + 1 = y_0 * 2 + 1 \wedge a = a_0\\
&\overline{\mathsf{wprec}}_1(S,P) = && = a > 0 \wedge y * 2 = y_0 * 2 \wedge y = y_0 \wedge y * 2 + 1 = y_0 * 2 + 1 \wedge a = a_0
\end{aligned}
$$

Fig. 3. Propagated conditions for Program 8

to the definition. We will now see that using preconditions and postconditions *simultaneously* allows for a precise identification of removable statements.

5.1. Removable Commands

We start by generalizing Lemma 1. This lemma states that there exist two equivalent ways to calculate verification conditions for a given program and specification: one based on weak preconditions and another based on strong postconditions. We will now see that for a given program this can be generalized: one can equally generate verification conditions by breaking the sequence of commands at any point, resulting in a prefix and a suffix of the initial command. The set of verification conditions is given as the union of the verification conditions of the suffix (computed using weak preconditions) and of the prefix (using strong postconditions). An additional verification condition relates the strong postcondition of the prefix and the weak precondition of the suffix. The first point in the following lemma formalizes this idea:

Lemma 5. Let (P, Q) be a specification and $S = C_1 \,;\, \ldots \,;\, C_n$ a program.

1. $\models \mathsf{VCG}^w(P,\, S,\, Q)$ iff $\models \overline{\mathsf{VC}}^s[k](S, P),\ \overline{\mathsf{spost}}_k(S, P) \rightarrow \overline{\mathsf{wprec}}_{k+1}(S, Q),\ \overline{\mathsf{VC}}^w[k+1](S, Q)$, for $k \in \{0, \ldots, n\}$

2. If $C_k = \textbf{if}\ b\ \textbf{then}\ S_t\ \textbf{else}\ S_f$ for some $k \in \{1, \ldots, n\}$, then

$$\begin{aligned}\models \mathsf{VCG}^w(P,\, S,\, Q) \quad \text{iff} \quad & \models \overline{\mathsf{VC}}^s[k-1](S, P),\ \mathsf{VCG}^w(\overline{\mathsf{spost}}_{k-1}(S, P) \wedge b,\, S_t,\, \overline{\mathsf{wprec}}_{k+1}(S, Q)),\\ & \mathsf{VCG}^w(\overline{\mathsf{spost}}_{k-1}(S, P) \wedge \neg b,\, S_f,\, \overline{\mathsf{wprec}}_{k+1}(S, Q)),\ \overline{\mathsf{VC}}^w[k+1](S, Q)\end{aligned}$$

3. If $C_k = \textbf{while}\ b\ \textbf{do}\ \{I\}\ S_b$ for some $k \in \{1, \ldots, n\}$, then

$$\begin{aligned}\models \mathsf{VCG}^w(P,\, S,\, Q) \quad \text{iff} \quad & \models \overline{\mathsf{VC}}^s[k-1](S, P),\ \overline{\mathsf{spost}}_{k-1}(S, P) \rightarrow I,\ \mathsf{VCG}^w(I \wedge b,\, S_b,\, I),\\ & I \wedge \neg b \rightarrow \overline{\mathsf{wprec}}_{k+1}(S, Q),\ \overline{\mathsf{VC}}^w[k+1](S, Q)\end{aligned}$$

Proof. 1. Applying repeatedly the definitions of wprec, VC^w, spost, VC^s, and Lemma 1:

$$\begin{aligned}& \models P \rightarrow \mathsf{wprec}(S, Q),\ \mathsf{VC}^w(S, Q)\\ \text{iff}\ & \models P \rightarrow \mathsf{wprec}(C_1, \mathsf{wprec}(C_2 \,;\, \ldots \,;\, C_n, Q)),\ \mathsf{VC}^w(C_1, \mathsf{wprec}(C_2 \,;\, \ldots \,;\, C_n, Q)),\ \mathsf{VC}^w(C_2 \,;\, \ldots \,;\, C_n, Q)\\ \text{iff}\ & \models \mathsf{VC}^s(C_1, P),\ \mathsf{spost}(C_1, P) \rightarrow \mathsf{wprec}(C_2 \,;\, \ldots \,;\, C_n, Q),\ \mathsf{VC}^w(C_2 \,;\, \ldots \,;\, C_n, Q)\\ \text{iff}\ & \models \mathsf{VC}^s(C_1, P),\ \mathsf{spost}(C_1, P) \rightarrow \mathsf{wprec}(C_2, \mathsf{wprec}(C_3 \,;\, \ldots \,;\, C_n, Q)),\ \mathsf{VC}^w(C_2, \mathsf{wprec}(C_3 \,;\, \ldots \,;\, C_n, Q)),\\ & \quad \mathsf{VC}^w(C_3 \,;\, \ldots \,;\, C_n, Q)\\ \text{iff}\ & \models \mathsf{VC}^s(C_1, P),\ \mathsf{VC}^s(C_2, \mathsf{spost}(C_1, P)),\ \mathsf{spost}(C_2, \mathsf{spost}(C_1, P)) \rightarrow \mathsf{wprec}(C_3 \,;\, \ldots \,;\, C_n, Q),\\ & \quad \mathsf{VC}^w(C_3 \,;\, \ldots \,;\, C_n, Q)\\ \text{iff}\ & \models \mathsf{VC}^s(C_1 \,;\, C_2, P),\ \mathsf{spost}(C_1 \,;\, C_2, P) \rightarrow \mathsf{wprec}(C_3 \,;\, \ldots \,;\, C_n, Q),\ \mathsf{VC}^w(C_3 \,;\, \ldots \,;\, C_n, Q)\\ & \ldots\\ \text{iff}\ & \models \mathsf{VC}^s(C_1 \,;\, \ldots \,;\, C_k, P),\ \mathsf{spost}(C_1 \,;\, \ldots \,;\, C_k, P) \rightarrow \mathsf{wprec}(C_{k+1} \,;\, \ldots \,;\, C_n, Q),\ \mathsf{VC}^w(C_{k+1} \,;\, \ldots \,;\, C_n, Q)\end{aligned}$$

2. Using the definition of the VCGen and Lemma 5 (1), one has the following

$$\begin{aligned}
&\models \mathsf{VCG}^w(P,\, S,\, Q)\\
\text{iff}\ \ &\models \overline{\mathsf{VC}}^s[k-1](S,P),\ \overline{\mathsf{spost}}_{k-1}(S,P) \rightarrow \overline{\mathsf{wprec}}_k(S,Q),\ \overline{\mathsf{VC}}^w[k](S,Q)\\
\text{iff}\ \ &\models \overline{\mathsf{VC}}^s[k-1](S,P),\\
&\quad \overline{\mathsf{spost}}_{k-1}(S,P) \rightarrow (b \rightarrow \mathsf{wprec}(S_t, \overline{\mathsf{wprec}}_{k+1}(S,Q))) \wedge (\neg b \rightarrow \mathsf{wprec}(S_f, \overline{\mathsf{wprec}}_{k+1}(S,Q))),\\
&\quad \mathsf{VC}^w(S_t, \overline{\mathsf{wprec}}_{k+1}(S,Q)),\ \mathsf{VC}^w(S_f, \overline{\mathsf{wprec}}_{k+1}(S,Q)),\ \overline{\mathsf{VC}}^w[k+1](S,Q)\\
\text{iff}\ \ &\models \overline{\mathsf{VC}}^s[k-1](S,P),\ \overline{\mathsf{spost}}_{k-1}(S,P) \wedge b \rightarrow \mathsf{wprec}(S_t, \overline{\mathsf{wprec}}_{k+1}(S,Q)),\ \mathsf{VC}^w(S_t, \overline{\mathsf{wprec}}_{k+1}(S,Q)),\\
&\quad \overline{\mathsf{spost}}_{k-1}(S,P) \wedge \neg b \rightarrow \mathsf{wprec}(S_f, \overline{\mathsf{wprec}}_{k+1}(S,Q)),\ \mathsf{VC}^w(S_f, \overline{\mathsf{wprec}}_{k+1}(S,Q)),\ \overline{\mathsf{VC}}^w[k+1](S,Q)\\
\text{iff}\ \ &\models \overline{\mathsf{VC}}^s[k-1](S,P),\ \mathsf{VCG}^w(\overline{\mathsf{spost}}_{k-1}(S,P) \wedge b,\, S_t,\, \overline{\mathsf{wprec}}_{k+1}(S,Q)),\\
&\quad \mathsf{VCG}^w(\overline{\mathsf{spost}}_{k-1}(S,P) \wedge \neg b,\, S_f,\, \overline{\mathsf{wprec}}_{k+1}(S,Q)),\ \overline{\mathsf{VC}}^w[k+1](S,Q)
\end{aligned}$$

3. We reason as follows, again using the definition of the VCGen and Lemma 5 (1)

$$\begin{aligned}
&\models \mathsf{VCG}^w(P,\, S,\, Q)\\
\text{iff}\ \ &\models \overline{\mathsf{VC}}^s[k-1](S,P),\ \overline{\mathsf{spost}}_{k-1}(S,P) \rightarrow \overline{\mathsf{wprec}}_k(S,Q),\ \overline{\mathsf{VC}}^w[k](S,Q)\\
\text{iff}\ \ &\models \overline{\mathsf{VC}}^s[k-1](S,P),\ \overline{\mathsf{spost}}_{k-1}(S,P) \rightarrow I,\ I \wedge b \rightarrow \mathsf{wprec}(S_b, I),\ I \wedge \neg b \rightarrow \overline{\mathsf{wprec}}_{k+1}(S,Q),\\
&\quad \mathsf{VC}^w(S_b, I),\ \overline{\mathsf{VC}}^w[k+1](S,Q)\\
\text{iff}\ \ &\models \overline{\mathsf{VC}}^s[k-1](S,P),\ \overline{\mathsf{spost}}_{k-1}(S,P) \rightarrow I,\ \mathsf{VCG}^w(I \wedge b,\, S_b,\, I),\ I \wedge \neg b \rightarrow \overline{\mathsf{wprec}}_{k+1}(S,Q),\\
&\quad \overline{\mathsf{VC}}^w[k+1](S,Q)
\end{aligned}$$

□

The significance of the first point of this lemma is that, according to the following proposition, it can be decided when the sequence $C_i\,;\,\ldots\,;\,C_j$ can be removed by considering the prefix $C_1\,;\,\ldots\,;\,C_{i-1}$ and the suffix $C_{j+1}\,;\,\ldots\,;\,C_n$.

Proposition 1. Let (P,Q) be a specification, $S = C_1\,;\,\ldots\,;\,C_n$ a program, and i, j, integers such that $1 \leq i \leq j \leq n$.

$$\text{If} \quad \models \overline{\mathsf{spost}}_{i-1}(S,P) \rightarrow \overline{\mathsf{wprec}}_{j+1}(S,Q) \quad \text{then} \quad \mathsf{remove}(i,j,S) \lhd_{(P,Q)} S$$

Proof. $\mathsf{remove}(i,j,S)$ is clearly a portion of S. Now let us assume that $\models \mathsf{VCG}^w(P,\, S,\, Q)$; we need to prove that $\models \mathsf{VCG}^w(P,\, \mathsf{remove}(i,j,S),\, Q)$. Applying Lemma 5 (1) to S with $k = i-1$ and $k = j$ we get respectively:

$$\begin{aligned}
&\models \mathsf{VCG}^w(P,\, S,\, Q) \quad \text{iff} \quad \models \overline{\mathsf{VC}}^s[i-1](S,P),\ \overline{\mathsf{spost}}_{i-1}(S,P) \rightarrow \overline{\mathsf{wprec}}_i(S,Q),\ \overline{\mathsf{VC}}^w[i](S,Q)\\
&\models \mathsf{VCG}^w(P,\, S,\, Q) \quad \text{iff} \quad \models \overline{\mathsf{VC}}^s[j](S,P),\ \overline{\mathsf{spost}}_{j}(S,P) \rightarrow \overline{\mathsf{wprec}}_{j+1}(S,Q),\ \overline{\mathsf{VC}}^w[j+1](S,Q)
\end{aligned}$$

Thus $\models \overline{\mathsf{VC}}^s[i-1](S,P)$ and $\models \overline{\mathsf{VC}}^w[j+1](S,Q)$. Now it suffices to apply Lemma 5 (1) to the program $\mathsf{remove}(i,j,S)$ with $k = i-1$. Since $\mathsf{remove}(i,j,S) = C_1\,;\,\ldots\,;\,C_{i-1}\,;\,C_{j+1}\,;\,\ldots\,;\,C_n$, this yields the following, which we apply from right to left.

$$\models \mathsf{VCG}^w(P,\, \mathsf{remove}(i,j,S),\, Q) \quad \text{iff} \quad \models \overline{\mathsf{VC}}^s[i-1](S,P),\ \overline{\mathsf{spost}}_{i-1}(S,P) \rightarrow \overline{\mathsf{wprec}}_{j+1}(S,Q),\ \overline{\mathsf{VC}}^w[j+1](S,Q)$$

□

We remark that since $\models \mathsf{VCG}^w(P,\, S,\, Q)$ implies $\models \overline{\mathsf{spost}}_{i-1}(S,P) \rightarrow \overline{\mathsf{wprec}}_i(S,Q)$ and $\models \overline{\mathsf{spost}}_{j}(S,P) \rightarrow \overline{\mathsf{wprec}}_{j+1}(S,Q)$, the following also hold:

$$\text{If} \quad \models \overline{\mathsf{wprec}}_i(S,Q) \rightarrow \overline{\mathsf{wprec}}_{j+1}(S,Q) \quad \text{then} \quad \mathsf{remove}(i,j,S) \lhd_{(P,Q)} S \tag{1}$$

$$\text{If} \quad \models \overline{\mathsf{spost}}_{i-1}(S,P) \rightarrow \overline{\mathsf{spost}}_{j}(S,P) \quad \text{then} \quad \mathsf{remove}(i,j,S) \lhd_{(P,Q)} S \tag{2}$$

However, note that the latter conditions are both stronger than the one in the proposition, which means that using them as tests could fail to identify some removable subprograms. This is in accordance with the

examples in Section 3, which have shown that simply propagating P forward and Q backward, and checking for implications between the propagated $\overline{\mathsf{spost}}_k(S,P)$ and then for implications between the propagated $\overline{\mathsf{wprec}}_k(S,Q)$, while sound, might result in slices that are not minimal. The method proposed in the literature calculates slices using these stronger tests, and this is the reason why they fail, for instance in Program 6. To illustrate our point with the latter program it suffices to note that since $\models \overline{\mathsf{spost}}_2(S,P) \rightarrow \overline{\mathsf{wprec}}_4(S,Q)$, the command C_3 can be removed according to the test of Proposition 1.

Proposition 1 in fact provides us with the *weakest* condition for slicing programs, since if we assume the initial program to be correct with respect to the given specification the reverse implication also holds:

Proposition 2. Let (P,Q) be a specification, $S = C_1\,;\,\ldots\,;\,C_n$ a program such that $\models \mathsf{VCG}^w(P,\,S,\,Q)$, and i, j, integers such that $1 \leq i \leq j \leq n$.

$$\text{If} \quad \mathsf{remove}(i,j,S) \lhd_{(P,Q)} S \quad \text{then} \quad \models \overline{\mathsf{spost}}_{i-1}(S,P) \rightarrow \overline{\mathsf{wprec}}_{j+1}(S,Q)$$

Proof. It suffices to prove that $\mathsf{VCG}^w(P,\,\mathsf{remove}(i,j,S),\,Q)$ implies $\models \overline{\mathsf{spost}}_{i-1}(S,P) \rightarrow \overline{\mathsf{wprec}}_{j+1}(S,Q)$. Again applying Lemma 5 (1) to $\mathsf{remove}(i,j,S)$ with $k = i-1$ yields the following, which we now apply from left to right.

$$\models \mathsf{VCG}^w(P,\,\mathsf{remove}(i,j,S),\,Q) \quad \text{iff} \quad \models \overline{\mathsf{VC}}^s[i-1](S,P),\ \overline{\mathsf{spost}}_{i-1}(S,P) \rightarrow \overline{\mathsf{wprec}}_{j+1}(S,Q),\ \overline{\mathsf{VC}}^w[j+1](S,Q)$$

□

This is a guarantee that our test identifies all removable subsequences of commands of a correct program – note that implications (1) and (2) cannot be reversed in this way.

Finally, note that the results in this section apply equally in the scope of total correctness and termination-sensitive slicing (the proofs are similar, using Lemma 2 instead of Lemma 1). In particular, Lemma 5 (1) and (2) hold with VCG^w_t (resp. $\overline{\mathsf{VC}}^w_t$, $\overline{\mathsf{VC}}^s_t$) substituted for VCG^w (resp. $\overline{\mathsf{VC}}^w$, $\overline{\mathsf{VC}}^s$). (3) is stated as follows: if $C_k = \textbf{while}\ b\ \textbf{do}\,\{I, e_v\}\,S_b$ for some $k \in \{1,\ldots,n\}$, then

$$\models \mathsf{VCG}^w_t(P,S,Q) \quad \text{iff} \quad \models \overline{\mathsf{VC}}^s_t[k-1](S,P),\ \overline{\mathsf{spost}}_{k-1}(S,P) \rightarrow I,\ I \wedge b \rightarrow e_v \geq 0,$$
$$\mathsf{VCG}^w_t(I \wedge b \wedge e_v = x_0, S_b, I \wedge e_v < x_0),\ I \wedge \neg b \rightarrow \overline{\mathsf{wprec}}_{k+1}(S,Q),\ \overline{\mathsf{VC}}^w_t[k+1](S,Q)$$

Proposition 1 applies with $\cdot \blacktriangleleft_{(P,Q)} \cdot$ substituted for $\cdot \lhd_{(P,Q)} \cdot$ (note that removing any subsequence from a terminating sequence of commands cannot result in a non-terminating sequence, so this is not surprising). Proposition 2 also applies, with VCG^w_t substituted for VCG^w.

5.2. Slicing Subprograms

The conditional branching and loop commands are structurally composed of sequences of commands. We will call these sequences (but not their subsequences) subprograms of the program under consideration. Given a specification (P,Q) for a program, we associate to each of its subprograms a local specification, which is obtained by propagating P and Q.

Definition 4 (Subprogram and Local Specification). Let S be a program and (P,Q) a specification for it. We will write $(\hat{P},\hat{S},\hat{Q}) \Subset (P,S,Q)$ with the meaning that $\hat{S}$ is a *subprogram* of S, and moreover given the specification (P,Q) for S the corresponding *local specification* of $\hat{S}$ is $(\hat{P},\hat{Q})$. The $\Subset$ relation is defined inductively as follows.

- $(P,S,Q) \Subset (P,S,Q)$;
- If $(P_1,S_1,Q_1) \Subset (P,S,Q)$ and $(P_2,S_2,Q_2) \Subset (P_1,S_1,Q_1)$ then $(P_2,S_2,Q_2) \Subset (P,S,Q)$.
- If $S = C_1\,;\,\ldots\,;\,C_n$ and $C_i = \textbf{if}\ b\ \textbf{then}\ S_t\ \textbf{else}\ S_f$ for some i with $1 \leq i \leq n$, then
  - $(\overline{\mathsf{spost}}_{i-1}(S,P) \wedge b, S_t, \overline{\mathsf{wprec}}_{i+1}(S,Q)) \Subset (P,S,Q)$
  - $(\overline{\mathsf{spost}}_{i-1}(S,P) \wedge \neg b, S_f, \overline{\mathsf{wprec}}_{i+1}(S,Q)) \Subset (P,S,Q)$

- If $S = C_1 ; \ldots ; C_n$ and $C_i = \textbf{while}\ b\ \textbf{do}\,\{I\}\,S_b$ for some i with $1 \leq i \leq n$, then $(I \wedge b, S_b, I) \Subset (P, S, Q)$.

As expected, subprograms of a correct program are correct with respect to their local specifications.

Lemma 6. Let S, $\hat{S}$ be programs and (P, Q) a specification such that $(\hat{P}, \hat{S}, \hat{Q}) \Subset (P, S, Q)$, i.e. $\hat{S}$ is a subprogram of S with local specification $(\hat{P}, \hat{Q})$. If $\models \mathsf{VCG}^w(P,\, S,\, Q)$ then $\models \mathsf{VCG}^w(\hat{P},\, \hat{S},\, \hat{Q})$.

Proof. By induction on the definition of the $\Subset$ relation. The first two cases are trivial. If $S = C_1 ; \ldots ; C_n$ and $C_i = \textbf{if}\ b\ \textbf{then}\ S_t\ \textbf{else}\ S_f$ for some $i \in \{1, \ldots, n\}$, Lemma 5 (2) yields $\models \mathsf{VCG}^w(\overline{\mathsf{spost}}_{i-1}(S, P) \wedge b,\, S_t,\, \overline{\mathsf{wprec}}_{i+1}(S, Q))$ and $\models \mathsf{VCG}^w(\overline{\mathsf{spost}}_{i-1}(S, P) \wedge \neg b,\, S_f,\, \overline{\mathsf{wprec}}_{i+1}(S, Q))$, as desired. Finally, if $S = C_1 ; \ldots ; C_n$ and $C_i = \textbf{while}\ b\ \textbf{do}\,\{I\}\,S_b$ for some $i \in \{1, \ldots, n\}$, Lemma 5 (3) yields $\models \mathsf{VCG}^w(I \wedge b,\, S_b,\, I)$. □

Let us now consider how the subprograms of S can be sliced with respect to a specification. The following proposition states that slicing a subprogram of a program with respect to its local specification results in a slice of the program.

Proposition 3. Let $S, \hat{S}$ be programs such that $(\hat{P}, \hat{S}, \hat{Q}) \Subset (P, S, Q)$. Moreover let $\hat{S}'$ be a portion of $\hat{S}$, i.e. $\hat{S}' \preceq \hat{S}$, and S' the program that results from replacing $\hat{S}$ by $\hat{S}'$ in S.

1. If $\hat{S}' \lhd_{(\hat{P},\hat{Q})} \hat{S}$ then $S' \lhd_{(P,Q)} S$
2. If $\models \mathsf{VCG}^w(P,\, S,\, Q)$ and $S' \lhd_{(P,Q)} S$ then $\hat{S}' \lhd_{(\hat{P},\hat{Q})} \hat{S}$

Proof. (1) Clearly S' must be a portion of S. The refinement aspect is proved by induction on the definition of $\Subset$ as follows.

- If $\hat{S} = S$ the result holds trivially, with $S' = \hat{S}'$.
- Let $(P_1, S_1, Q_1) \Subset (P, S, Q)$ and $(\hat{P}, \hat{S}, \hat{Q}) \Subset (P_1, S_1, Q_1)$. Moreover let S_1' be the program that results from replacing $\hat{S}$ by $\hat{S}'$ in S_1. Then by induction hypothesis one has that $\hat{S}' \lhd_{(\hat{P},\hat{Q})} \hat{S}$ implies $S_1' \lhd_{(P_1,Q_1)} S_1$. Now let S' denote the result of replacing S_1 by S_1' in S. Then again by induction hypothesis one has that $S_1' \lhd_{(P_1,Q_1)} S_1$ implies $S' \lhd_{(P,Q)} S$. Note that S' can also be seen as the result of replacing $\hat{S}$ by $\hat{S}'$ in S, thus we are done since $\hat{S}' \lhd_{(\hat{P},\hat{Q})} \hat{S}$ implies $S' \lhd_{(P,Q)} S$.
- Let $S = C_1 ; \ldots ; C_{i-1} ; \textbf{if}\ b\ \textbf{then}\ \hat{S}\ \textbf{else}\ S_f ; C_{i+1} ; \ldots ; C_n$. We assume $\models \mathsf{VCG}^w(P,\, S,\, Q)$; using Lemma 5 (2), this implies

$$\models \overline{\mathsf{VC}}^s[i-1](S, P),\ \mathsf{VCG}^w(\overline{\mathsf{spost}}_{i-1}(S, P) \wedge b,\, \hat{S},\, \overline{\mathsf{wprec}}_{i+1}(S, Q)),$$
$$\mathsf{VCG}^w(\overline{\mathsf{spost}}_{i-1}(S, P) \wedge \neg b,\, S_f,\, \overline{\mathsf{wprec}}_{i+1}(S, Q)),\ \overline{\mathsf{VC}}^w[i+1](S, Q)$$

  and since $\hat{S}' \lhd_{(\overline{\mathsf{spost}}_{i-1}(S,P) \wedge b,\, \overline{\mathsf{wprec}}_{i+1}(S,Q))} \hat{S}$, this in turn implies

$$\models \overline{\mathsf{VC}}^s[i-1](S, P),\ \mathsf{VCG}^w(\overline{\mathsf{spost}}_{i-1}(S, P) \wedge b,\, \hat{S}',\, \overline{\mathsf{wprec}}_{i+1}(S, Q)),$$
$$\mathsf{VCG}^w(\overline{\mathsf{spost}}_{i-1}(S, P) \wedge \neg b,\, S_f,\, \overline{\mathsf{wprec}}_{i+1}(S, Q)),\ \overline{\mathsf{VC}}^w[i+1](S, Q)$$

  Now observe that since $S' = C_1 ; \ldots ; C_{i-1} ; \textbf{if}\ b\ \textbf{then}\ \hat{S}'\ \textbf{else}\ S_f ; C_{i+1} ; \ldots ; C_n$, again by Lemma 5 (2) one has $\models \mathsf{VCG}^w(P,\, S',\, Q)$. The case when $\hat{S}$ is the *else* branch is similar.
- Let $S = C_1 ; \ldots ; C_{i-1} ; \textbf{while}\ b\ \textbf{do}\,\{I\}\,\hat{S} ; C_{i+1} ; \ldots ; C_n$. We assume $\models \mathsf{VCG}^w(P,\, S,\, Q)$; using Lemma 5 (3), this implies

$$\models \overline{\mathsf{VC}}^s[i-1](S, P),\ \overline{\mathsf{spost}}_{i-1}(S, P) \rightarrow I,\ \mathsf{VCG}^w(I \wedge b,\, \hat{S},\, I),\ I \wedge \neg b \rightarrow \overline{\mathsf{wprec}}_{i+1}(S, Q),\ \overline{\mathsf{VC}}^w[i+1](S, Q)$$

  and since $\hat{S}' \lhd_{(I \wedge b, I)} \hat{S}$, this in turn implies

$$\models \overline{\mathsf{VC}}^s[i-1](S, P),\ \overline{\mathsf{spost}}_{i-1}(S, P) \rightarrow I,\ \mathsf{VCG}^w(I \wedge b,\, \hat{S}',\, I),\ I \wedge \neg b \rightarrow \overline{\mathsf{wprec}}_{i+1}(S, Q),\ \overline{\mathsf{VC}}^w[i+1](S, Q)$$

  Now observe that since $S' = C_1 ; \ldots ; C_{i-1} ; \textbf{while}\ b\ \textbf{do}\,\{I\}\,\hat{S}' ; C_{i+1} ; \ldots ; C_n$, again by Lemma 5 (3) one has $\models \mathsf{VCG}^w(P,\, S',\, Q)$.

(2) $\hat{S}'$ is a portion of $\hat{S}$; for the refinement aspect it suffices to prove that $\models \mathsf{VCG}^w(P, S', Q)$ implies $\mathsf{VCG}^w(\hat{P}, \hat{S}', \hat{Q})$. Again this is proved by induction on the definition of $\Subset$. We illustrate this for the conditional case, with $S = C_1 ; \ldots ; C_{i-1} ;$ **if** b **then** $\hat{S}$ **else** $S_f ; C_{i+1} ; \ldots ; C_n$. We have by Lemma 5 (2) that

$$\models \overline{\mathsf{VC}}^s[i-1](S,P),\ \mathsf{VCG}^w(\overline{\mathsf{spost}}_{i-1}(S,P) \wedge b,\ \hat{S}',\ \overline{\mathsf{wprec}}_{i+1}(S,Q)),$$
$$\mathsf{VCG}^w(\overline{\mathsf{spost}}_{i-1}(S,P) \wedge \neg b,\ S_f,\ \overline{\mathsf{wprec}}_{i+1}(S,Q)),\ \overline{\mathsf{VC}}^w[i+1](S,Q)$$

and thus $\models \mathsf{VCG}^w(\overline{\mathsf{spost}}_{i-1}(S,P) \wedge b,\ \hat{S}',\ \overline{\mathsf{wprec}}_{i+1}(S,Q))$. $\square$

Recall that terminating programs admit non-terminating programs as specification-based slices, since termination-insensitive slicing merely forces the preservation of the loop invariants. In particular, since $\mathbf{skip} \triangleleft_{(I \wedge b, I)} S$ always holds (because $\models I \wedge b \rightarrow I$), for any P, Q, and I one has that **while** b **do** $\{I\}$ **skip** $\triangleleft_{(P,Q)}$ **while** b **do** $\{I\}$ S. Consequently, any program admits as a slice the program that results from removing the body of every loop.

Of course, this does not apply in a total correctness setting. Again, if one substitutes $\cdot \blacktriangleleft_{(P,Q)} \cdot$ for $\cdot \triangleleft_{(P,Q)} \cdot$ and VCG_t^w for VCG^w, the previous results are valid also in the context of termination-sensitive slicing (the proofs are similar). It suffices to consider an extra case in the definition of subprogram, for loops annotated with variants, as follows:

- If $S = C_1 ; \ldots ; C_n$ and $C_i =$ **while** b **do** $\{I, e_v\}$ S_b for some i with $1 \leq i \leq n$, then $(I \wedge b \wedge e_v = x_0, S_b, I \wedge e_v < x_0) \Subset (P, S, Q)$.

The proposition can be used to further slice a program by slicing its subprograms with respect to their local specifications. Conditional branches are sliced by propagating the postcondition inside both branches, as well as the precondition strengthened with the boolean condition and its negation, respectively. In the case of a loop with invariant I and condition b, it suffices to use as specification for the body of the loop the assertions $(I \wedge b, I)$, or $(I \wedge b \wedge e_v = x_0, I \wedge e_v < x_0)$ for termination-sensitive slicing.

This can be of use in two scenarios mentioned in Section 4. If the program is being sliced to remove redundant code, using a specification with respect to which it has been proved correct, then the loop annotations were adequate to prove correctness, and the proposition allows the removal of redundant code to proceed inside loops (each loop body can be sliced with respect to the preservation of its invariant and the strict decrease of its variant).

In a specialization / reuse scenario, the program is being sliced based on a weaker specification than the one with respect to which it was originally proved correct. In this scenario, it may well be the case that the loop invariants annotated in the program are stronger than they need to be – they have been used to establish correctness with respect to a stronger specification than the one now being used. In order to allow the slicing process to proceed usefully inside loop bodies, the user should first replace the invariants by weaker versions that are sufficient for proving correctness with respect to the new specification, and only then use Proposition 3 to slice the loop bodies with these weaker invariants.

Slicing Criterion. The above discussion raises a question concerning the criterion to be used for comparing the quality of different slices of the same program. The criterion that was implicit in Section 3 considered the total number of commands or lines of code in a slice. Note that for programs consisting only of atomic commands (i.e. for sequences of **skip** and assignment commands) the number of commands and lines (assuming one command per line) are the same. In the presence of commands containing subprograms however, this is not so, since our program syntax dictates that a loop or a conditional is a single command, regardless of the length of their body / branch subprograms. A more appropriate measure (i.e. closer to the notion of "lines of code") is the number of *atomic commands and boolean conditions*.

Based on this criterion, selecting the minimal slice of a given program $S = C_1 ; \ldots ; C_n$ implies taking into consideration the number of commands and boolean conditions of each subprogram $\hat{S}$ of S – it makes no sense to just count the number (between 1 and n) of top-level commands in each slice of S. Moreover, since each $\hat{S}$ can be sliced with respect to its local specification following Proposition 3, the general structure of a slicing algorithm should be to slice S only after slicing each of its subprograms with respect to its local specification. Only then can the minimal slice of S be selected.

5.3. Intermediate Conditions

A major difference between the notions of slicing based on assertions and traditional notions based on dependencies is that in the latter the slicing criterion always includes a *line number* k; in forward slicing one asks for instructions not dependent on the values at line k of a given set of variables to be removed; in backward slicing it is the instructions on which the values of the variables at line k do not depend that are removed.

This can be mimicked in our context by introducing an *intermediate assertion* to be taken into account for calculating a slice. We briefly explain here how our framework can be extended in this direction.

Definition 5 (Specification-based Slice with Intermediate Condition). Let $S = C_1 ; \ldots ; C_n$ be a program, (P, Q) a specification, and R an assertion such that $\models \mathsf{spost}(C_1 ; \ldots ; C_k, P) \rightarrow R$ and $\models R \rightarrow \mathsf{wprec}(C_{k+1} ; \ldots ; C_n, Q)$. We say that the program $S' = S'_1 ; S'_2$ is a slice of S with respect to the specification (P, Q) and intermediate condition R at position k, written $S' \lhd_{(P,R,k,Q)} S$, if

$$S'_1 \lhd_{(P,R)} C_1 ; \ldots ; C_k \quad \text{and} \quad S'_2 \lhd_{(R,Q)} C_{k+1} ; \ldots ; C_n$$

Although Definition 5 is sufficient to illustrate the idea, it can be generalized so that the intermediate condition regards some subprogram of S. Multiple intermediate conditions can also be admitted.

Naturally, such slices are particular cases of specification-based slices – the intermediate assertion simply restricts the definition further with respect to the specification. The following lemma is straightforward to prove using Lemma 5 (1).

Lemma 7. If $S' \lhd_{(P,R,k,Q)} S$ then $S' \lhd_{(P,Q)} S$.

Intermediate assertions can be used with the practical goal of facilitating automatic proofs by inserting conditions that are obviously true at the given line, which may allow more commands to be sliced off. But they also enrich the power of specification-based slicing, allowing one to slice fragments of the code with respect to local conditions, possibly even omitting part of the global specification. An extreme case is to compute a slice consisting of a postcondition-based slice of a prefix of a program, followed by a precondition-based slice of a suffix, as in $S' \lhd_{(\mathsf{wprec}(C_1 ; \ldots ; C_k, R), R, k, \mathsf{spost}(C_{k+1} ; \ldots ; C_n, R))} S$. In this case the intermediate condition is the only slicing criterion considered.

6. Labeled Control Flow Graphs

In Section 3 we have shown that the quadratic time algorithms – even our improved version, which always eliminates the longest contiguous subsequence of S – do not produce minimal slices. In Section 7 we will show that the problem of selecting commands to be removed from a program in order to produce a minimal slice can be formulated as a graph problem, and solved using standard graph algorithms. In this section we set up a basis for this, by defining a notion of assertion-labeled control flow graph of a program with respect to a given specification.

Definition 6 (Labeled Control Flow Graph). Given a program S, precondition P and postcondition Q such that $S = C_1 ; \ldots ; C_n$, the *labeled control flow graph* $LCFG(S, P, Q)$ of S with respect to (P, Q) is a labeled directed acyclic graph (WDAG) whose edge labels are pairs of logical assertions on program states. To each command C in program S we associate its input node $IN(C)$ and its ouput node $OUT(C)$ in the graph $LCFG(S, P, Q)$. The graph is constructed as follows:

1. Each command C_i in S will be represented by one (in the case of **skip** and assignment commands) or two nodes (for conditional and loop commands).
   - If C_i is **skip** or an assignment command, let there be a new node C_i in the graph. We set $IN(C_i) = OUT(C_i) = C_i$.
   - If $C_i =$ **if** b **then** S_t **else** S_f, let there be two new nodes $if(b)$ and fi in the graph. We set $IN(C_i) = if(b)$ and $OUT(C_i) = fi$.
   - If $C_i =$ **while** b **do** $\{I\}\, S$ or $C_i =$ **while** b **do** $\{I, e_v\}\, S$, let there be two new nodes $do(b)$ and od in the graph. We set $IN(C_i) = do(b)$ and $OUT(C_i) = od$.

2. Let $LCFG(S, P, Q)$ also contain two additional nodes $START$ and END.

3. Let $LCFG(S, P, Q)$ contain 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 $(OUT(C_n), END)$. The labels of these edges are set as follows

$$\begin{aligned} label(START, IN(C_1)) &= (\overline{\mathsf{spost}}_0(S, P), \overline{\mathsf{wprec}}_1(S, Q)) = (P, \overline{\mathsf{wprec}}_1(S, Q)); \\ label(OUT(C_i), IN(C_{i+1})) &= (\overline{\mathsf{spost}}_i(S, P), \overline{\mathsf{wprec}}_{i+1}(S, Q)) \quad \text{for } i \in \{1, \ldots, n-1\}; \\ label(OUT(C_n), END) &= (\overline{\mathsf{spost}}_n(S, P), \overline{\mathsf{wprec}}_{n+1}(S, Q)) = (\overline{\mathsf{spost}}_n(S, P), Q). \end{aligned}$$

4. For $i \in \{1, \ldots, n\}$, if $C_i = \textbf{if } b \textbf{ then } S_t \textbf{ else } S_f$, we recursively construct the graphs

$$LCFG(S_t, b \wedge \overline{\mathsf{spost}}_{i-1}(S, P), \overline{\mathsf{wprec}}_{i+1}(S, Q)) \quad \text{and} \quad LCFG(S_f, \neg b \wedge \overline{\mathsf{spost}}_{i-1}(S, P), \overline{\mathsf{wprec}}_{i+1}(S, Q))$$

   These graphs are grafted into the present graph by removing their $START$ nodes and setting the origin of the dangling edges to be in both cases the node $IN(C_i)$, and similarly removing their END nodes and setting the destination of the dangling edges to be the node $OUT(C_i)$.

5. For $i \in \{1, \ldots, n\}$, if $C_i = \textbf{while } b \textbf{ do } \{I\}\, S$, we recursively construct the graph

$$LCFG(S, I \wedge b, I)$$

   If a loop variant is present, with $C_i = \textbf{while } b \textbf{ do } \{I, e_v\}\, S$, we construct instead the graph

$$LCFG(S, I \wedge b \wedge e_v = x_0, I \wedge e_v < x_0)$$

   This graph is grafted into the present graph by removing its $START$ node and setting the origin of the dangling edge to be the node $IN(C_i)$, and similarly removing its END node and setting the destination of the dangling edge to be the node $OUT(C_i)$.

Clearly every subprogram $\hat{S}$ of S is represented by a subgraph of $LCFG(S, P, Q)$ delimited by a pair of nodes $START/END$, if/fi, or do/od. Let us denote these nodes respectively by $IN(\hat{S})$ and $OUT(\hat{S})$. The basic intuition of labeled CFGs is that for every pair of consecutive commands $\hat{C}_i$, $\hat{C}_{i+1}$ in $\hat{S}$, there exists an edge $(\hat{C}_i, \hat{C}_{i+1})$ in $LCFG(S, P, Q)$ whose label consists of the strong postcondition of the prefix of $\hat{S}$ ending with $\hat{C}_i$, and the weak precondition of the suffix of $\hat{S}$ beginning with $\hat{C}_{i+1}$, with respect to the local specification $(\hat{P}, \hat{Q})$ of $\hat{S}$ propagated from (P, Q). If $\models \mathsf{VCG}^w(\hat{P},\, \hat{S},\, \hat{Q})$ then every edge in the subgraph representing $\hat{S}$ has a label (ϕ, ψ) such that $\models \phi \rightarrow \psi$, as a consequence of Lemma 5 (1). Moreover, by Lemma 6, if $\models \mathsf{VCG}^w(P,\, S,\, Q)$ then this must be true for every subprogram $\hat{S}$ of S, and thus every edge in the graph $LCFG(S, P, Q)$ has a label (ϕ, ψ) such that $\models \phi \rightarrow \psi$.

If loops are annotated with variants, this is taken into account when constructing the subgraph corresponding to the loop's body (point 5 of the definition). So we now have that $\models \mathsf{VCG}_t^w(P, S, Q)$ implies $\models \phi \rightarrow \psi$ for every label (ϕ, ψ) in the graph.

Finally, we remark that the LCFG of a program can be constructed in three steps by first building the unlabeled CFG from the syntax tree of the program; then assigning the first component of the labels by traversing the graph from $START$ to END; and finally assigning the second component by traversing the graph in the reverse direction. In each of these traversals the label of each edge can be calculated *locally* from the labels of the (one or two) previous edges. In particular, for $1 \leq k \leq n$ we have $\overline{\mathsf{spost}}_k(S, P) = \mathsf{spost}(C_k, \overline{\mathsf{spost}}_{k-1}(S, P))$ and $\overline{\mathsf{wprec}}_k(S, Q) = \mathsf{wprec}(C_k, \overline{\mathsf{wprec}}_{k+1}(S, Q))$. Note however that the cost of constructing the graph is not linear on the program size, since weak preconditions are potentially of exponential size on the length of the program (but this can be corrected to quadratic, see Section 8).

In this paper, labeled control flow graphs are used as a basis for the definition of slice graphs in the next section. We observe however that they are interesting entities on their own; in [dCHP] we explore their application for the interactive generation of verification conditions.

7. Slice Graphs

We will now define a notion of *slice graph*, in which removable sequences of commands are associated with edges added to the initial control flow graph.

Definition 7 (Slice Graph). Consider a program S and a specification (P, Q) such that $\models \mathsf{VCG}^w(P,\ S,\ Q)$ (in which case we assume loops are not annotated with variants) or $\models \mathsf{VCG}_t^w(P, S, Q)$. The *slice graph* $SLCG(S, P, Q)$ of S with respect to (P, Q) is obtained from the labeled control flow graph $LCFG(S, P, Q)$ by inserting additional edges as follows.

For every subprogram $\hat{S} = \hat{C}_1 ; \ldots ; \hat{C}_n$, with $(\hat{P}, \hat{S}, \hat{Q}) \Subset (P, S, Q)$,

- If $\models \hat{P} \rightarrow \hat{Q}$, a new **skip** node is inserted in the graph, together with two edges $(IN(\hat{S}), \mathbf{skip})$ and $(\mathbf{skip}, OUT(\hat{S}))$, both with label $(\hat{P}, \hat{Q})$.
- For all $j \in \{1, \ldots, n\}$ if $\models \hat{P} \rightarrow \overline{\mathsf{wprec}}_{j+1}(\hat{S}, \hat{Q})$, an edge $(IN(\hat{S}), IN(\hat{C}_{j+1}))$ with label $(\hat{P}, \overline{\mathsf{wprec}}_{j+1}(\hat{S}, \hat{Q}))$ is inserted;
- For all $i \in \{1, \ldots, n\}$, if $\models \overline{\mathsf{spost}}_{i-1}(\hat{S}, \hat{P}) \rightarrow \hat{Q}$, an edge $(OUT(\hat{C}_{i-1}), OUT(\hat{S}))$ with label $(\overline{\mathsf{spost}}_{i-1}(\hat{S}, \hat{P}), \hat{Q})$ is inserted;
- For all $i, j \in \{1, \ldots, n\}$ such that $i < j$, if $\models \overline{\mathsf{spost}}_{i-1}(\hat{S}, \hat{P}) \rightarrow \overline{\mathsf{wprec}}_{j+1}(\hat{S}, \hat{Q})$, an edge $(OUT(\hat{C}_{i-1}), IN(\hat{C}_{j+1}))$ with label $(\overline{\mathsf{spost}}_{i-1}(\hat{S}, \hat{P}), \overline{\mathsf{wprec}}_{j+1}(\hat{S}, \hat{Q}))$ is inserted;

Note that this construction is purely based on the LCFG of S: $\hat{P}$, $\hat{Q}$, $\overline{\mathsf{spost}}_{i-1}(\hat{S}, \hat{P})$, and $\overline{\mathsf{wprec}}_{j+1}(\hat{S}, \hat{Q})$ can be read from labels of edges in the subgraph corresponding to the subprogram being considered. For each subprogram, first-order conditions are generated for every pair of edges such that the first precedes the second in the graph (the order in which this is done is irrelevant). If the validity of some condition cannot be established, the corresponding edge will not be added to the graph, in accordance with the requirement that slicing must be conservative.

As an example, Figure 4 partially shows the slice graph for program 5 with respect to the specification $(y > 10, x \geq 0)$. Removable sequences are signaled by the thick edges (and one **skip** node) that are added to the initial labeled CFG. Many edges are omitted to lighten the presentation of the graph; two edges are missing in the *then* branch (from $x := 100$ to $x := x - 100$ and from $x := x + 50$ to *fi*), and in the *else* branch five edges are missing – note that since the first component of every label in this path is a contradiction, an edge is inserted from each node to every reachable node in the *else* branch).

For any given subprogram $\hat{S}$ of S, the slice graph contains as subgraph the LCFG of every slice of $\hat{S}$ with respect to its local specification, and consequently also the LCFG of the program that results from replacing in S any subprogram by one if its slices. The following result formalizes this fact.

Lemma 8. In the conditions of Definition 7, let i, j be integers such that $1 \leq i \leq j \leq n$, and S' the program resulting from replacing $\hat{S}$ by $\mathsf{remove}(i, j, \hat{S})$. Then

1. If $\models \mathsf{VCG}^w(P,\ S,\ Q)$,

$$\mathsf{remove}(i, j, \hat{S}) \lhd_{(\hat{P},\hat{Q})} \hat{S} \quad \text{iff} \quad \text{the graph } LCFG(S', P, Q) \text{ is a subgraph of } SLCG(S, P, Q).$$

2. If $\models \mathsf{VCG}_t^w(P, S, Q)$,

$$\mathsf{remove}(i, j, \hat{S}) \blacktriangleleft_{(\hat{P},\hat{Q})} \hat{S} \quad \text{iff} \quad \text{the graph } LCFG(S', P, Q) \text{ is a subgraph of } SLCG(S, P, Q).$$

Proof. We prove 1 (the proof of 2 is similar, since propositions 1 and 2 also apply for termination-sensitive slicing, as explained at the end of Section 5.1).
(Only if part) By Proposition 2, $\models \overline{\mathsf{spost}}_{i-1}(\hat{S}, \hat{P}) \rightarrow \overline{\mathsf{wprec}}_{j+1}(\hat{S}, \hat{Q})$. Clearly the graph $LCFG(S', P, Q)$ is equal to $LCFG(S, P, Q)$, with the exception of a subgraph that is no longer present, and is replaced by an edge (or two edges and a **skip** node). Moreover these new edges are present in the graph $SLCG(S, P, Q)$, following Definition 7.

(If part) We prove that $\models \mathsf{VCG}^w(\hat{P},\ \mathsf{remove}(i, j, \hat{S}),\ \hat{Q})$ (note that $\models \mathsf{VCG}^w(\hat{P},\ \hat{S},\ \hat{Q})$ must hold, following Lemma 6). The graph $LCFG(S', P, Q)$ is the same as $LCFG(S, P, Q)$, except for the subgraphs corresponding

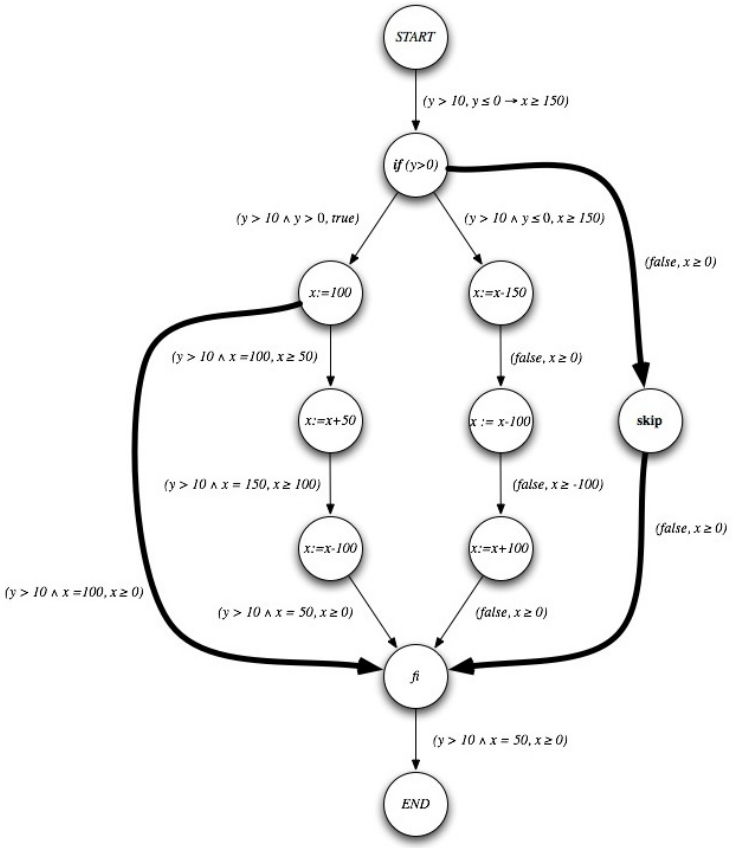

Fig. 4. Example slice graph (extract). Thick lines represent edges that were added to the initial CFG, corresponding to "shortcut" subprograms that do not modify the semantics of the program. These paths have the same origin and destination nodes as other longer paths corresponding to removable sequences

to the commands removed inside $\hat{S}$. If $LCFG(S', P, Q)$ is a subgraph of $SLCG(S, P, Q)$ then the edge or edges that short-circuit the subgraph corresponding to the removed commands can only have been introduced (by Definition 7) because $\models \overline{\mathsf{spost}}_{i-1}(\hat{S}, \hat{P}) \rightarrow \overline{\mathsf{wprec}}_{j+1}(\hat{S}, \hat{Q})$, and Proposition 1 allows us to conclude the proof. □

A consequence of the previous result is that all slices of a program are represented in its slice graph, and no other portions of S are.

Proposition 4. Let S, S' be programs such that $S' \preceq S$. Then

1. If $\models \mathsf{VCG}^w(P, S, Q)$,

   $S' \triangleleft_{(P,Q)} S$ iff the control flow graph $LCFG(S', P, Q)$ is a subgraph of $SLCG(S, P, Q)$.

2. If $\models \mathsf{VCG}^w_t(P, S, Q)$,

   $S' \blacktriangleleft_{(P,Q)} S$ iff the control flow graph $LCFG(S', P, Q)$ is a subgraph of $SLCG(S, P, Q)$.

Proof. We prove 1 (the proof of 2 is similar, since Proposition 3 also applies for termination-sensitive slicing, as explained in Section 5.2).
(Only if part) The slice S' is a portion of S, which is obtained by removing some top-level commands of S and doing this recursively for some of the subprograms of S. By Proposition 3 (2), all the commands removed in every subprogram of S must result in slices regarding the local specification of the corresponding

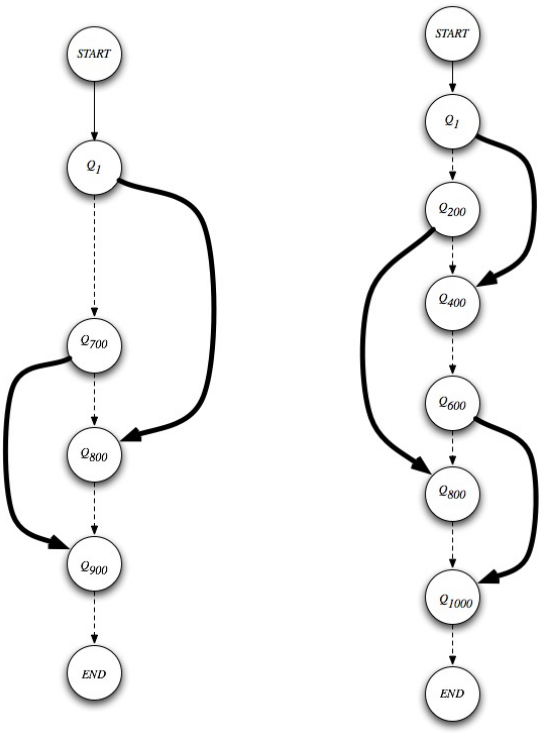

Fig. 5. Example slice graphs

subprogram. Consider any sequencing of these subprogram slicing operations; the proof proceeds by induction on the length of this sequence using Lemma 8 (from left to right).
(If part) It is clear that in any control-flow graph that is a subgraph of $SLCG(S, P, Q)$, each edge (or pair of edges with a **skip** node) that is not present in the initial graph $LCFG(S, P, Q)$ has been added relative to a certain subprogram of S and short-circuits some commands in that subprogram. We consider any sequencing of these extra edges; the proof proceeds by induction on the length of this sequence, using Lemma 8 (from right to left) and Proposition 3 (1). □

The slice graph then represents the entire set of specification-based slices of S, and obtaining the minimal slice is simply a matter of selecting the shortest subsequences using the information in the graph.

Slicing Algorithms. A consequence of the previous result is that the problem of determining the minimal slice of a given program S with respect to the specification (P, Q) can be reduced to determining the minimal control flow graph contained in the slice graph $G = SLCG(S, P, Q)$.

Consider the case of programs without loops or conditionals, consisting only of atomic commands. Figure 5 shows the slice graphs for the two problematic examples presented in Section 3. It is clear that for such programs the control flow graph of the minimal slice (i.e. the slice containing the smallest number of atomic commands) can be determined by a standard (unweighted) shortest paths algorithm (basically a breadth-first traversal, linear on the size of the graph). This CFG contains of course a single path from *START* to *END*.

For programs containing loops, the same algorithm can be used. Following our remarks on slicing subprograms in Section 5, determining a minimal slice of a program implies determining the minimal slices of its subprograms, but from the point of view of slice graphs this is irrelevant: when facing a *while* loop, the shortest path algorithm will have to cross from the *do* node to the *od* node, and will naturally determine the minimal slice of the loop body subprogram.

Conditional commands pose a more substantial problem. Simply applying a shortest paths algorithm would select *one of the branches* of the conditional; what is required is to slice both branches with respect to their local specifications, and then take into account the total number of lines of code of both branches,

when slicing the sequence of commands containing this conditional. We sketch one way to do this combining a *weighted* shortest paths algorithm with graph rewriting, as follows:

1. Assign a weight of 1 to every edge of the slice graph G.
2. For all conditional commands that do not contain any conditional commands as subprograms,
   (i) run a shortest paths algorithm on the subgraphs of G corresponding to both branches of the conditional, and let $x = 1 + l + r$, where l and r are the sum of the weights of the resulting paths in the *then* and *else* branch respectively;
   (ii) replace both these subgraphs by *a single edge* with origin *if* and destination *fi*, with weight x;
3. More conditional commands containing no branching in their subprogram graphs may now have been created; repeat from step 2.

Some mechanism must additionally be used to keep track of the rewritten subgraphs, to allow the slice to be read back from the final graph.

Finally, note that the notion of minimality implicit in this discussion is relative, since it is meant with respect to a slice graph: the proof tool may have failed or timed out in checking some valid condition, and thus an edge that should have been included in the graph is missing; the resulting slice will only be as good as the graph.

Intermediate Assertions. Computing slices in the presence of intermediate assertions as introduced in Section 5.3 requires no major modifications in our setting. It suffices to locate in the slice graph the edge (C_k, C_{k+1}) with label (ϕ, ψ), and replace it by two new edges (C_k, New) and (New, C_{k+1}) with labels (ϕ, R) and (R, ψ) respectively, where *New* is a new node inserted in the graph. The standard algorithm will then compute a slice taking the intermediate condition into consideration.

8. Conclusion

Our online laboratory [dCHP10] incorporates all the slicing algorithms discussed in Section 3, as well as our algorithm based on slice graphs.[5] The laboratory can also be used for program verification; in particular it allows the user to generate verification conditions in an interactive way and includes visualization capabilities, see [dCHP]. The latter paper also shows that the notion of control flow graph labeled with semantic information is of independent interest for program verification.

While the front-end is meant to allow for experimentation and comparison of different algorithms, one obligatory step will be to calculate weak preconditions using Flanagan and Saxe's algorithm [FS01], which avoids the potential exponential explosion in the size of the conditions generated, keeping our algorithm within quadratic time. We intend also to explore alternatives to strong postcondition calculations, to eliminate the use of existential quantifiers. One such alternative is the notion of *update*, as used prominently in the dynamic logic of the KeY system [ABB$^+$05].

In future work we intend to construct a slicer for an intermediate verification language like BoogiePL [BCD$^+$05]. This is a language used by a number of different verification tools for realistic languages, which are first translated into the intermediate language to take advantage of a common VCGen. We foresee that a slicer for this language could very easily cope with code of the different high-level languages that can be translated into it.

Acknowledgment. This work was supported by the projects RESCUE (PTDC/EIA/65862/2006), FAVAS (PTDC/EIA-CCO/105034/2008), and CROSS (PTDC/EIA-CCO/108995/2008), all funded by Fundação para a Ciência e Tecnologia (FCT).

[5] Available from `http://gamaepl.di.uminho.pt/gamaslicer`.

References

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4(1):32–54, 2005.

[BCC+05] 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.

[BCD+05] 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.

[BCF+10] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA LIST and INRIA, 2010.

[BdCHP10] 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 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM'10)*, pages 93–102. IEEE Computer Society, 2010.

[BRLS04] 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, March 2004.

[CCL98] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–608, November 1998. Special issue on program slicing.

[CH96] Joseph J. Comuzzi and Johnson M. Hart. Program slicing using weakest preconditions. In *FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, pages 557–575, London, UK, 1996. Springer-Verlag.

[CLYK01] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 605–609, New York, NY, USA, 2001. ACM.

[dCHP] Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Verification graphs for programs with contracts. Submitted for publication.

[dCHP10] Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Gamaslicer: an Online Laboratory for Program Verification and Analysis. In *LDTA '10: Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, pages 1–8, New York, NY, USA, 2010. ACM.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[FDHH01] Chris Fox, Sebastian Danicic, Mark Harman, and Robert M. Hierons. Backward conditioning: A new program specialisation technique and its application to program comprehension. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC'01)*, pages 89–97. IEEE Computer Society, 2001.

[FP11] Maria João Frade and Jorge Sousa Pinto. Verification Conditions for Source-level Imperative Programs. *Computer Science Review*, 5:252–277, 2011.

[FS01] 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.

[HHF+01] Mark Harman, Robert M. Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 138–147. IEEE Computer Society, 2001.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10), 1992.

[War09] Martin Ward. Properties of slicing definitions. In *SCAM '09: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 23–32, Washington, DC, USA, 2009. IEEE Computer Society.

[Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[XQZ+05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.