Assertion-based Slicing and Slice Graphs

José Bernardo Barros, Daniela da Cruz, Pedro Rangel Henriques, Jorge Sousa Pinto
Departamento de Informática / CCTC
Universidade do Minho
Braga, Portugal
{*jbb,danieladacruz,prh,jsp*}*@di.uminho.pt*

***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. 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). The paper also reviews in detail, through examples, the ideas behind the use of preconditions and postconditions for slicing programs.**

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

I. Introduction

Program slicing [1] is a well-established activity in software engineering. For instance 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 spurious 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.

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 [2] 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 [3], 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) [4]; Spec# [5], a formal language for C# API contracts; and the ANSI/ISO C Specification Language (ACSL) [6].

There are several points of contact between slicing and verification: first, traditional syntactic slicing, applied a priori, facilitates the verification of large programs. Secondly, and this is what concerns us in this paper, it makes sense to slice programs based on semantic, rather than syntactic, criteria, and the contracts used in DbC and program verification are excellent candidates for such criteria. A third point (see Section VII) is that there is evidence that this kind of slicing can also be of help in the verification of large programs.

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* [7], a form of forward slicing. This was shown to subsume both static and dynamic notions of syntax-based slicing, since the initial state of execution is constrained by a first-order formula that can be used to constrain the set of different 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.

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 spurious with respect to the specification (P', Q').

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, and introduces 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 claim that our algorithm produces minimal slices (in a sense that will be made precise)

Structure of the Paper: Section II introduces the simple language considered in the paper, and the definitions of weakest precondition and strongest postcondition. In Section III 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, and in Section IV we introduce formally the definitions used in the rest of the paper. The next sections introduce a new test for identifying removable chunks of code (Section V) and a graph-based algorithm for actually computing the best possible slices of a program with respect to a given specification (Section VI). We conclude the paper in Section VII.

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

Figure 1. Language syntax

$$
\begin{array}{rcl}
\mathsf{wp}(\mathbf{skip}, Q) & = & Q \\
\mathsf{wp}(x := e, Q) & = & Q^x_e \\
\mathsf{wp}(C_1; C_2, Q) & = & \mathsf{wp}(C_1, \mathsf{wp}(C_2, Q)) \\
\mathsf{wp}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, Q) & = & (b \rightarrow \mathsf{wp}(C_t, Q)) \\
 & & \wedge\ (\neg b \rightarrow \mathsf{wp}(C_f, Q)) \\
\mathsf{wp}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\, C, Q) & = & I \\
 & & \\
\mathsf{sp}(\mathbf{skip}, P) & = & P \\
\mathsf{sp}(x := e, P) & = & \exists v.\, P^x_v \wedge x == e^x_v \\
\mathsf{sp}(C_1; C_2, P) & = & \mathsf{sp}(C_2, \mathsf{sp}(C_1, P)) \\
\mathsf{sp}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, P) & = & \mathsf{sp}(C_t, b \wedge P) \\
 & & \vee\ \mathsf{sp}(C_f, \neg b \wedge P) \\
\mathsf{sp}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\, C, Q) & = & I \wedge \neg b
\end{array}
$$

Figure 2. Definition of weakest precondition and strongest postcondition. Q^x_e denotes the result of substituting e for x in Q; I is a loop invariant.

II. The Language, WP and SP

To illustrate our ideas we use the syntax in Figure 1 for a core imperative language. Programs are non-empty sequences of commands and specifications are programs annotated with preconditions and postconditions.

We remark that the choice of language is not important, and the ideas discussed scale up to realistic languages; the only crucial requirements are the existence of an axiomatic semantics (definitions of weakest precondition and strongest postcondition), and an external proof tool that is capable of reasoning about the data structures that are present in the language. To illustrate our ideas we use a very simple language with integer variables only; the syntax of assertions (used as preconditions, postconditions, and loop invariants) is obtained as an extension of boolean expressions with first-order quantification.

The notions of weakest precondition and strongest postcondition are certainly among the most important and popular in programming semantics. For such a simple language, there is a nice symmetry between them, and both can be used to calculate proof obligations (usually called *verification conditions*) when verifying the correctness of programs. The former is however much more widely used in verification tools, because of the absence of quantifiers. The definition of both notions for our language is given in Figure 2.

Notation: Let $S = C_1 ; \ldots ; C_n$, $1 \leq k \leq n$, and $1 \leq i \leq j \leq n$. We will use the following notation for the weakest precondition of a suffix of S; the strongest postcondition of a prefix of S; and the sequence obtained by removing a subsequence of S.

- $\overline{\mathsf{wp}}_k(S, Q) \doteq \mathsf{wp}(C_k ; C_{k+1} ; \ldots ; C_n, Q)$
- $\overline{\mathsf{wp}}_{n+1}(S, Q) \doteq Q$
- $\overline{\mathsf{sp}}_0(S, P) \doteq P$
- $\overline{\mathsf{sp}}_k(S, P) \doteq \mathsf{sp}(C_1 ; \ldots ; C_{k-1} ; C_k, P)$
- $\mathsf{remove}(i, j, S) \doteq$
  $\begin{cases} \textbf{skip} \text{ if } i = 1 \text{ and } j = n, \\ C_1 ; \ldots ; C_{i-1} ; C_{j+1} ; \ldots ; C_n \text{ otherwise.} \end{cases}$

III. Related work

In this section we discuss the existent notions of slicing based on the preconditions and postconditions of a program, as well as algorithms that calculate them. The notions of *postcondition-based slice* and *precondition-based slice* (and the closed related notion of *conditioned slice*) were introduced independently. It is important to distinguish between these definitions (which admit many slices for the same program and specification) and the corresponding slices calculated by specific algorithms (in particular those introduced in the same papers). But first we refer to semantic forms of slicing not based on assertions.

A. 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 [8] 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. In the rest of this section we review work on assertion-based slicing.

B. Postcondition-based Slicing

The idea of slicing programs based on their specifications was introduced by Comuzzi et al. [9] 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.

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

Program 1.

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

Program 2.

Consider for instance Program 1. The postcondition $Q = x \geq 0$ yields the weakest precondition $x \geq -50$. If the program is executed in a state in which this precondition holds (i.e. it is a correct program with respect to the given postcondition), 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 weakest 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 $\models \overline{\mathsf{wp}}_i(S, Q) \rightarrow Q$, the sequence $C_i ; \ldots ; C_n$ can be removed. For the previous example we have $\overline{\mathsf{wp}}_3(S, Q) = x \geq 100$, $\overline{\mathsf{wp}}_2(S, Q) = x \geq 50$, and $\overline{\mathsf{wp}}_1(S, Q) = x \geq -50$. Now observe that $\models \overline{\mathsf{wp}}_2(S, Q) \rightarrow Q$, 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.

P-slices are of course *not unique*. For instance since $\models \overline{\mathsf{wp}}_3(S, Q) \rightarrow Q$ 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 highest number of instructions are removed.

It is important to understand that not only suffixes of a sequence of commands may be removed. Consider the postcondition $Q = x \geq 0$ for Program 2. Calculating the weakest preconditions we have $\overline{\mathsf{wp}}_3(S, Q) = x \geq -100$, $\overline{\mathsf{wp}}_2(S, Q) = x \geq -200$, and $\overline{\mathsf{wp}}_1(S, Q) = x \geq -50$ (the weakest precondition of the program with respect to $x \geq 0$). Note that although $\not\models \overline{\mathsf{wp}}_1(S, Q) \rightarrow Q$, the commands in lines 1 and 2 can be removed because $\models \overline{\mathsf{wp}}_1(S, Q) \rightarrow \overline{\mathsf{wp}}_3(S, Q)$. Thus the postcondition will be preserved.

A Simple Linear Time Algorithm: It is easy to produce p-slices. It suffices to perform a traversal of the syntax tree, generating for each node a first-order proof obligation of the form $\models Q' \rightarrow Q$, which is then used to decide if the current node can be sliced off. If the proof obligation can be discharged by some automatic proof tool, then the entire tree whose root is the current node is sliced off; otherwise the algorithm conservatively proceeds to the node's descendants and tries to slice at a deeper level. This algorithm runs in

linear time on the length of the program, as long as a time-out limit is considered for the prover to discard obligations.

Problems of the Linear Time Algorithm: It is easy to understand from programs 1 and 2 that the same program may contain removable prefixes, suffixes, and even removable sequences that are neither prefixes nor suffixes. Moreover, these removable sequences may well overlap. Thus it is clear that no linear traversal of the syntax tree can possibly detect all removable sequences: the traversal dictates the slice that is calculated without any concern for minimality.

Better algorithms for computing p-slices of a command sequence $S = C_1 ; \ldots ; C_n$ can be synthesized by iterating the following basic step that removes the subsequence $C_i ; \ldots ; C_j$, with $1 \leq i \leq j \leq n$:

- If $\models \overline{\mathsf{wp}}_i(S, Q) \rightarrow \overline{\mathsf{wp}}_{j+1}(S, Q)$ then replace S by $C_1 ; \ldots ; C_{i-1} ; C_{j+1} ; \ldots ; C_n$.

Original Quadratic Time Algorithm: The algorithm proposed by Comuzzi runs in *quadratic time* on the length of the sequence. The algorithm first tries to slice the entire program by removing its longest removable suffix, and then repeats this task, considering successively shorter prefixes of the resulting program, and removing their longest removable suffixes. For instance in a program with 10 statements it would consider in this order the sequences $(1..10), (2..10), \ldots, (10..10), (1..9), (2..9), \ldots, (9..9), (1..8)$, and so on. For each sequence a proof obligation (implication involving two weakest preconditions) is generated; if it can be proved then the sequence is removed.

The Quadratic Algorithm Fails to Remove the Longest Sequence: Take a sequence of 1000 statements such that $\models \overline{\mathsf{wp}}_1(S, Q) \rightarrow \overline{\mathsf{wp}}_{800}(S, Q)$ and $\models \overline{\mathsf{wp}}_{700}(S, Q) \rightarrow \overline{\mathsf{wp}}_{900}(S, Q)$. Two subsequences may be sliced off, consisting respectively of commands 1 to 799 and 700 to 899. The algorithm will consider the shorter sequence first, and in doing so will eliminate the possibility of the longer sequence being removed, since line 800 will be removed (and it may happen that $\overline{\mathsf{wp}}_1(S, Q)$ is not stronger than any remaining $\overline{\mathsf{wp}}_k(S, Q)$). 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 to be sliced. Thus in the 10 statement program one would consider sequences in the order $(1..10), (1..9), (2..10), (1..8), (2..9), (3..10), (1..7), \ldots$ This would certainly remove the longest removable sequence.

The Improved Quadratic Algorithm Is Not Optimal: Consider the case in which $\models \overline{\mathsf{wp}}_1(S, Q) \rightarrow \overline{\mathsf{wp}}_{400}(S, Q)$, $\models \overline{\mathsf{wp}}_{600}(S, Q) \rightarrow \overline{\mathsf{wp}}_{1000}(S, Q)$, and $\models \overline{\mathsf{wp}}_{200}(S, Q) \rightarrow \overline{\mathsf{wp}}_{800}(S, Q)$. The longest sequence will be sliced off (600 program lines), but this will preclude the possibility of slicing two shorter sequences that would together consist of 800 program lines. So overall removing the larger contiguous sequence may not produce the smallest slice. In fact it should now be clear that considering all sequences in any given order cannot result in a minimal slice. The same is true for postcondition-based and specification-based slices, discussed below. In Section VI we will show that this problem can in general be formulated as a graph problem.

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

Program 3.

C. Precondition-based Slicing

Canfora et al. [7] later introduced *conditioned slicing*, which uses preconditions as a means to specify a set of initial states for computing a forward slice, in which unreachable code can be eliminated. The intention of the authors was to obtain a form of forward slicing that subsumes both *static* (all possible initial states considered) and *dynamic* slicing (for a single initial state, i.e. a single run of the program).

Chung and colleagues [10] later introduced *precondition-based slicing* as the dual notion of postcondition-based slicing. Precondition-based and conditioned slicing are very close notions, except that the latter is formalized in terms of symbolic execution, whereas the former is formalized in terms of strongest postconditions.

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, so in fact they can be sliced off. The remaining instructions in the program will still be executed with $x \geq 0$, whereas with the original program the instruction in line 3 was reached in a state characterized by the weaker $x \geq -100$. So whatever postcondition was true after execution of the initial program, it will still be true after execution of the sliced program.

To be more systematic, let the program $S = C_1 ; \ldots ; C_n$ be executed in a state in which the precondition P holds. Then for the above fragment we have $\overline{\mathsf{sp}}_1(S, P) = \exists v.v \geq 0 \wedge x = v + 100$, $\overline{\mathsf{sp}}_2(S, P) = \exists v.v \geq 0 \wedge x = v - 100$, and $\overline{\mathsf{sp}}_3(S, P) = \exists v.v \geq 0 \wedge x = v + 100$. We see that $\models P \rightarrow \overline{\mathsf{sp}}_2(S, P)$, 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 algorithm proposed in [10]). In the same example program, since in fact $\models \overline{\mathsf{sp}}_1(S, P) \rightarrow \overline{\mathsf{sp}}_3(S, P)$, we could alternatively slice off lines 2 and 3 of the program, which shows that removable sequences may overlap.

D. Specification-based Slicing

A *specification-based slice* can be calculated when both a precondition P and a postcondition Q are given for program S, i.e. S is supposed to be executed in initial states satisfying P, and if execution stops the final state is known to satisfy Q. Programs resulting from S by removing a set of statements,

```
1 if (y > 0) then x := 100;
2                 x := x+50;
3                 x := x−100
4            else x := x−150;
5                 x := x−100;
6                 x := x+100
```

Program 4.

```
1 if (y > 0) then x := 100
2            else skip
```

Program 5.

and for which this is still true, are said to be specification-based slices of S with respect to (P, Q).

The method proposed in [10] 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 4 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 the condition $y > 0$ and its negation respectively. In the second case this yields false, which will result in the *else* sequence being completely sliced off (since false implies anything). Postcondition-based slicing with $x \geq 0$ will then produce the sliced Program 5 ([10] advocates replacing the whole conditional command by the first branch, but it is debatable whether this transformation can still be called slicing).

But again it is very easy to show that the resulting slices are not minimal, as can be seen by looking at Program 6 with specification $(\mathsf{true}, x \geq 100)$. We have:

$$\begin{array}{rcl} P & = & \overline{\mathsf{sp}}_0(S,P) = \mathsf{true} \\ & & \overline{\mathsf{sp}}_1(S,P) = \exists v.x = v * v \\ & & \overline{\mathsf{sp}}_2(S,P) = \exists v.x = v * v + 100 \\ & & \overline{\mathsf{sp}}_3(S,P) = \exists v.x = v * v + 150 \end{array}$$

and

$$\begin{array}{l} \overline{\mathsf{wp}}_1(S,Q) = \mathsf{true} \\ \overline{\mathsf{wp}}_2(S,Q) = x \geq -50 \\ \overline{\mathsf{wp}}_3(S,Q) = x \geq 50 \\ \overline{\mathsf{wp}}_4(S,Q) = x \geq 100 = Q \end{array}$$

It is obvious that the postcondition is satisfied after execution of the instruction in line 2, which means that line 3 can be removed and 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 the spurious instruction, since no forward implications are valid among the $\overline{\mathsf{sp}}_i(S,P)$ or the $\overline{\mathsf{wp}}_i(S,Q)$. Composing both slicing algorithms will of course not solve this fundamental flaw: trying to identify removable statements using only preconditions or only postconditions may fail.

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

Program 6.

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

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

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

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

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

Figure 3. Definition of relation "is a portion of" on programs and commands

IV. Assertion-based Slices

We now formalize the notions of slicing reviewed in the previous section. A program S' is a *specification-based slice* of S if it is a *portion* of S (a syntactic notion, also known as a *reduction* 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 1 (Portion-of relation)**.** *The $\cdot \preceq \cdot$ relation is the partial order generated by the set of axioms and rules given in Figure 3.*

Definition 2 (Assertion-based slices)**.** *Let S be a correct program with respect to the 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 S' is also correct with respect to (P, Q);*
- *a* precondition-based slice *of S with respect to P if $S' \lhd_{(P,\mathsf{sp}(S,P))} S$;*
- *a* postcondition-based slice *of S with respect to postcondition Q if $S' \lhd_{(\mathsf{wp}(S,Q),Q)} S$.*

Incidentally, we remark that the names precondition- and postcondition-based slice are not entirely adequate for describing the notions known under these names. They would more accurately be described as *condition-based forward and backward slice*, respectively, which not only establishes a correspondence with the two classic approaches to syntactic slicing, but also highlights the fact that conditions may be propagated even when preconditions or postconditions are not present. An example is a program that starts with an assignment $x := c$ with c a constant. Even without an informative precondition, forward slicing will use the information $x = c$, calculated as the strongest postcondition of the command, and then propagated forward.

In this paper we propose a solution to the problems raised in the previous section, in the form of an optimal slicing algorithm. The algorithm builds on two basic ideas, that will be explained in the next two sections. In abstract terms, any slicing algorithm based on the axiomatic semantics of programs must be able to

1) Identify subprograms that can be removed from the program being sliced, without modifying its semantics. More concretely, given a program $S = C_1 ; \ldots ; C_n$ with specification (P, Q), some test is used to allow the algorithm to decide if $\mathsf{remove}(i, j, S) \lhd_{(P,Q)} S$ holds. We have shown that the algorithm of [10] fails to identify some subprograms; In Section V we will show that using preconditions and postconditions *simultaneously* allows for a precise identification of removable subprograms.
2) Select, among the subprograms identified as removable, the combination that produces the smallest sliced program. In Section VI it will be shown that this may be reduced to a graph problem that can be solved by applying standard algorithms.

The algorithm can be applied to calculate precondition-, postcondition-, and specification-based slices. We concentrate on the latter case, since the first two are particular cases as shown before.

Note that the resulting algorithm is *optimal in a relative sense* only. 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.

V. Removable Subprograms

We start by considering programs without iteration and postpone the discussion of loops to the end of the section. For such programs the following lemma is straightforward to prove:

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

1) $\models P \rightarrow \mathsf{wp}(S, \mathsf{sp}(S, P))$
2) $\models \mathsf{sp}(S, \mathsf{wp}(S, Q)) \rightarrow Q$
3) $\models P \rightarrow \mathsf{wp}(S, Q)$ *iff* $\models \mathsf{sp}(S, P) \rightarrow Q$

Each of the implications mentioned in the third item in fact corresponds to the *verification condition* for the program S: it suffices to check the validity of this condition to ensure that S is correct with respect to the specification (P, Q).

In this section we consider the following problem: given a specification and a program S correct with respect to it, how can it be decided if some subsequence of S can be removed, resulting in a program that is still correct with respect to the specification, i.e. it is a specification-based slice of S? Note that we are not asking if the sequence *should* be sliced (since this could prevent the minimal slice from being obtained); we leave that question to the next section.

The following lemma establishes the implications that are valid among the calculated preconditions and postconditions calculated for a subsequence of a given correct program.

Lemma 2. *Let (P, Q) be a specification; $S = C_1 ; \ldots ; C_n$ a program such that $\models P \rightarrow \mathsf{wp}(S, Q)$, and i, j, k integers such that $1 \leq i \leq j \leq n$ and $0 \leq k \leq n$. Then*

1) $\models \overline{\mathsf{sp}}_k(S, P) \rightarrow \overline{\mathsf{wp}}_{k+1}(S, Q)$
2) $\models \overline{\mathsf{sp}}_{i-1}(S, P) \rightarrow \mathsf{wp}(C_i ; \ldots ; C_j, \overline{\mathsf{wp}}_{j+1}(S, Q))$

Proof: 1 can be proved by induction from the correctness of S. 2 is a consequence of 1 and the observation that

$$\overline{\mathsf{wp}}_i(S, Q) = \mathsf{wp}(C_i ; \ldots ; C_j, \overline{\mathsf{wp}}_{j+1}(S, Q)) \text{ and}$$
$$\mathsf{sp}(C_i ; \ldots ; C_j, \overline{\mathsf{sp}}_{i-1}(S, P)) = \overline{\mathsf{sp}}_j(S, P)$$

■

Observe that $\overline{\mathsf{sp}}_{i-1}(S, P)$ and $\overline{\mathsf{wp}}_{j+1}(S, Q)$ can be seen respectively as the *strongest precondition* and the *weakest postcondition* calculated for the sequence $C_i ; \ldots ; C_j$ w.r.t. the specification (P, Q). Their significance is that, according to the following proposition, they can be used to decide exactly when the sequence $C_i ; \ldots ; C_j$ can be sliced off.

Proposition 1. *In the conditions of the previous lemma,*

$$\models \overline{\mathsf{sp}}_{i-1}(S, P) \rightarrow \overline{\mathsf{wp}}_{j+1}(S, Q) \text{ iff } \mathsf{remove}(i, j, S) \lhd_{(P,Q)} S$$

Proof: $\mathsf{remove}(i, j, S)$ is clearly a portion of S. Now applying repeatedly Lemma 1(3) and the definitions of weakest precondition and strongest postcondition one can prove

$$\models \mathsf{sp}(C_1 ; \ldots ; C_{i-1}, P) \rightarrow \mathsf{wp}(C_{j+1} ; \ldots ; C_n, Q) \quad \text{iff}$$
$$\models P \rightarrow \mathsf{wp}(\mathsf{remove}(i, j, S), Q)$$

■

We remark that the following are implications but not equivalences:

$$\models \overline{\mathsf{wp}}_i(S, Q) \rightarrow \overline{\mathsf{wp}}_{j+1}(S, Q) \text{ implies}$$
$$\models P \rightarrow \mathsf{wp}(\mathsf{remove}(i, j, S), Q)$$

$$\models \overline{\mathsf{sp}}_{i-1}(S, P) \rightarrow \overline{\mathsf{sp}}_j(S, P) \text{ implies}$$
$$\models P \rightarrow \mathsf{wp}(\mathsf{remove}(i, j, S), Q)$$

Both conditions would also imply $S' \lhd_{(P,Q)} S$. However, note that the latter conditions are both stronger than the one in the proposition (as a consequence of Lemma 2(1)), which means that using them as tests would not allow for all removable subprograms to be identified. This is in accordance with the examples in Section III, which have shown that simply propagating P forward and Q backward, and checking for implications between the propagated $\overline{\mathsf{sp}}_k(S,P)$ and then for implications between the propagated $\overline{\mathsf{wp}}_k(S,Q)$, while sound, may result in slices that are not minimal. We turn back to Program 6 to illustrate our point. Since $\models \overline{\mathsf{sp}}_2(S,P) \rightarrow \overline{\mathsf{wp}}_4(S,Q)$, the command $C_3 \equiv x := 50$ can be removed according to our test.

For commands containing sequences of commands, illustrated here with conditional, the following proposition states that slicing both branches results in a slice of the structured command. It suffices to propagate the postcondition inside both branches, as well as the precondition strengthened with the boolean condition and its negation, respectively.

Proposition 2. *If $S'_t \lhd_{(P \wedge b, Q)} S_t$ and $S'_f \lhd_{(P \wedge \neg b, Q)} S_f$, then*

$$\textbf{if } b \textbf{ then } S'_t \textbf{ else } S'_f \lhd_{(P,Q)} \textbf{if } b \textbf{ then } S_t \textbf{ else } S_f$$

Proof: The portion requisite is immediate, and by Proposition 1 we have $\models P \wedge b \rightarrow \mathsf{wp}(S'_t, Q)$ and $\models P \wedge \neg b \rightarrow \mathsf{wp}(S'_f, Q)$, from which it follows that:
$\models P \rightarrow ((b \rightarrow \mathsf{wp}(S'_t, Q)) \wedge (\neg b \rightarrow \mathsf{wp}(S'_f, Q)))$. ■

The treatment of loops introduces a few subtleties. First, if S contains loops the implication $\models P \rightarrow \mathsf{wp}(S,Q)$ is no longer the only verification condition: other conditions must be introduced related to the preservation of the *loop invariant*, as well as its relation with the loop's desired postcondition (or precondition, if strongest postconditions are used). The notion of refinement, required by the definition of specification-based slicing, will now incorporate the preservation of these additional conditions. Moreover, in a total correctness setting other conditions are involved, regarding the strictly decreasing value of a *loop variant*. Slicing the body of a terminating loop should not result in a non-terminating loop, which is granted by the preservation of the verification conditions involving the loop variant. Full details will be given in a long version of this paper.

VI. Slice Graphs

We define below a notion of control graph for a program, labeled with respect to a given specification, and the notion of slice graph, in which removable sequences of commands will be associated with edges added to the initial control flow graph.

Definition 3 (Labeled Control Flow Graph)**.** *Given a program S, precondition P and postcondition Q such that $S = C_1; \ldots; C_n$ and $\models P \rightarrow \mathsf{wp}(S,Q)$, the* labeled control flow graph *$LCFG(S,P,Q)$ of S with respect to (P,Q) is a weighted directed acyclic graph (WDAG) whose weights 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:*

- *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 labeled C_i in the graph. We set $IN(C_i) = OUT(C_i) = C_i$.*
  - *If $C_i = \textbf{if } b \textbf{ then } S_t \textbf{ else } S_f$, let there be two new nodes, labeled $C_i^{if(b)}$ and C_i^{fi} in the graph. We set $IN(C_i) = C_i^{if(b)}$ and $OUT(C_i) = C_i^{fi}$.*
  - *If $C_i = \textbf{while } b \textbf{ do } \{I\}\, S$, let there be two new nodes, labeled $C_i^{do(b)}$ and C_i^{od} in the graph. We set $IN(C_i) = C_i^{do(b)}$ and $OUT(C_i) = C_i^{od}$.*
- *Let $LCFG(S,P,Q)$ also contain two additional nodes labeled $START$ and END.*
- *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 weights of these edges are set as follows*

$$w(START, IN(C_1)) = (\overline{\mathsf{sp}}_0(S,P), \overline{\mathsf{wp}}_1(S,Q));$$
$$w(OUT(C_i), IN(C_{i+1})) = (\overline{\mathsf{sp}}_i(S,P), \overline{\mathsf{wp}}_{i+1}(S,Q)) \text{ for } i \in \{1,\ldots,n-1\};$$
$$w(OUT(C_n), END) = (\overline{\mathsf{sp}}_n(S,P), \overline{\mathsf{wp}}_{n+1}(S,Q)).$$

- *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 $LCFG(S_t, b \wedge \overline{\mathsf{sp}}_{i-1}(S,P), \overline{\mathsf{wp}}_{i+1}(S,Q))$ and $LCFG(S_f, \neg b \wedge \overline{\mathsf{sp}}_{i-1}(S,P), \overline{\mathsf{wp}}_{i+1}(S,Q))$. These graphs are grafted into the present graph by removing their $START$ nodes and setting the source 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 $OUT(C_i)$.*
- *For $i \in \{1,\ldots,n\}$, if $C_i = \textbf{while } b \textbf{ do } \{I\}\, S$, we construct the graph $LCFG(S, I \wedge b, I)$ recursively. This graph is grafted into the present graph by removing its $START$ node and setting the source 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 $OUT(C_i)$.*

Informally, the idea is that the weight of an edge $C_i \longrightarrow C_j$ represents the strongest postcondition $\overline{\mathsf{sp}}_i(S,P)$ of (the sequence ending with) the command C_i and the weakest precondition $\overline{\mathsf{wp}}_j(S,Q)$ of (the sequence beginning with) the command C_j, calculated from the initial specification (P,Q) taking into account the structure of the program.

Lemma 3. *Let $G = LCFG(S,P,Q)$, and $\hat{S} = \hat{C_1}\,;\,\ldots\,;\,\hat{C_m}$ any subprogram of S. Then for any i,j such that $1 \leq i \leq j \leq m$ and the edge $(OUT(\hat{C_i}), IN(\hat{C_j}))$ exists in G, we have $w(OUT(C_i), IN(C_j)) =$*
$(\mathsf{sp}(\hat{C_1}\,;\,\ldots\,;\,\hat{C_i}, \hat{P}), \mathsf{wp}(\hat{C_j}\,;\,\ldots\,;\,\hat{C_m}, \hat{Q}))$
where $\hat{P}$ is the first component of the weight of the incoming edge into the node $IN(\hat{C_1})$, and $\hat{Q}$ is the second component of the weight of the outgoing edge from the node $OUT(\hat{C_m})$.

Proof: It suffices to observe that the local definition of the weights (from the weights of neighbouring edges) corresponds to the definition of strongest postconditions and weakest preconditions. ∎

It is crucial that sequences that are branches of a conditional are generated using the appropriate strongest postcondition and weakest precondition, in accordance with Proposition 2. The same applies to the body of loop commands. Together with Lemma 3 this means that the graph is annotated exactly with the strongest postconditions and weakest preconditions that are calculated recursively throughout the structure of the graph, following the definition of Figure 2. The labelled CFG can thus be seen as a "verification graph" for a program; in particular, the program is correct if $\models P \rightarrow \overline{\mathsf{wp}}_1(S,Q)$, where $(P, \overline{\mathsf{wp}}_1(S,Q))$ is the weight of the outgoing edge from the $START$ node or equivalently if $\models \overline{\mathsf{sp}}_n(S,P) \rightarrow Q$, where $(\overline{\mathsf{sp}}_n(S,P), Q)$ is the weight of the incoming edge into the END node.

An algorithm for constructing the graph could first build the unweighted graph from the syntax tree of the program, then assign the first component of the weights by traversing the graph from $START$ to END, and finally assign the second component by traversing the graph in the reverse direction. Note that the weight of each edge can be calculated *locally* from the weights of the (one or two) previous edges. In particular, note that for $1 \leq k \leq n$,

$$\overline{\mathsf{sp}}_k(S,P) = \mathsf{sp}(C_k, \overline{\mathsf{sp}}_{k-1}(S,P))$$
$$\overline{\mathsf{wp}}_k(S,Q) = \mathsf{wp}(C_k, \overline{\mathsf{wp}}_{k+1}(S,Q))$$

The worst-case execution cost of constructing the graph is apparently linear on the program size. However, weakest preconditions are potentially of exponential size on the length of the program [11], so this is not so. Fortunately this size can be corrected to quadratic (see Section VII).

Definition 4 (Slice Graph)**.** *Given a program S, precondition P and postcondition Q such that $\models P \rightarrow \mathsf{wp}(S,Q)$, 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.*

Let $\hat{S} = \hat{C_1}\,;\,\ldots\,;\,\hat{C_m}$ be any maximal *sequence of commands in S, i.e. $\hat{S}$ is a branch of a conditional command in S, or the body of a loop command in S, or else $\hat{S} = S$. Then for any two edges $(\hat{C}_{i-1}, \hat{C}_i)$ with weight $(\overline{\mathsf{sp}}_{i-1}(S,P), \overline{\mathsf{wp}}_i(S,Q))$ and $(\hat{C}_j, \hat{C}_{j+1})$ with weight $(\overline{\mathsf{sp}}_j(S,P), \overline{\mathsf{wp}}_{j+1}(S,Q))$ in $LCFG(S,P,Q)$ such that $i < j$, if $\models \overline{\mathsf{sp}}_{i-1}(S,P) \rightarrow \overline{\mathsf{wp}}_{j+1}(S,Q)$,*

- *if $i \neq 1$ or $j \neq m$, an edge $(\hat{C}_{i-1}, \hat{C}_{j+1})$ with weight $(\overline{\mathsf{sp}}_{i-1}(S,P), \overline{\mathsf{wp}}_{j+1}(S,Q))$ is inserted;*
- *otherwise if $i = 1$ and $j = m$ a new* **skip** *node is inserted in the graph, together with two edges $(\hat{C}_{i-1}, \mathbf{skip})$ and $(\mathbf{skip}, \hat{C}_{j+1})$, both with weight $(\overline{\mathsf{sp}}_{i-1}(S,P), \overline{\mathsf{wp}}_{j+1}(S,Q))$.*

The time required to insert the additional edges into the graph is again quadratic on the length of the program, since for each sequence of commands it is necessary to generate slicing conditions for every pair of edges such that the first precedes the second in the graph. We remark that this presupposes that the external theorem prover checks the validity of formulas in constant time, which is a reasonable assumption since automatic tools are typically used with a time out limit, after which a condition is treated as invalid. Also, the construction depends on the particular external tool used to decide which edges should be inserted, and may in fact result in different graphs if different tools are used.

As an example, Figure 4 shows the slice graph for program 4 with respect to the specification $(y > 10, x \geq 0)$. It is clear that removable sequences are signaled by the edges (and possibly **skip** nodes) that are added to the initial labeled CFG. The following proposition states that all admissible slices are represented in the slice graph.

Proposition 3. *Let $S' \preceq S$. Then $S' \lhd_{(P,Q)} S$ iff the control flow graph $LCFG(S',P,Q)$ is a spanning subgraph (i.e. a subgraph with the same set of nodes) of the slice graph $SLCG(S,P,Q)$.*

Proof: (Sketch) First observe that by Lemma 3, Definition 4, and the subsequent observations, the weights of the edges in the slice graph $SLCG(S,P,Q)$ correspond to the preconditions and postconditions required by Propositions 1 and 2. If S' is a slice of S then the commands that have possibly been removed are described by those two propositions, so appropriate "short-circuit" edges have been inserted in the slice graph. Moreover, the CFG of S' must also contain those edges. For the reverse implication, we note that any spanning subgraph of the slice graph that is not the full graph will contain edges corresponding to removable sequences, and in fact the CFG of the graph obtained by actually removing those sequences must coincide with the said subgraph.

∎

Thus for any given sequence of commands σ inside S, the graph contains a path that represents every subsequence σ' of σ such that substituting σ' for σ results in a slice of S. The slice graph 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.

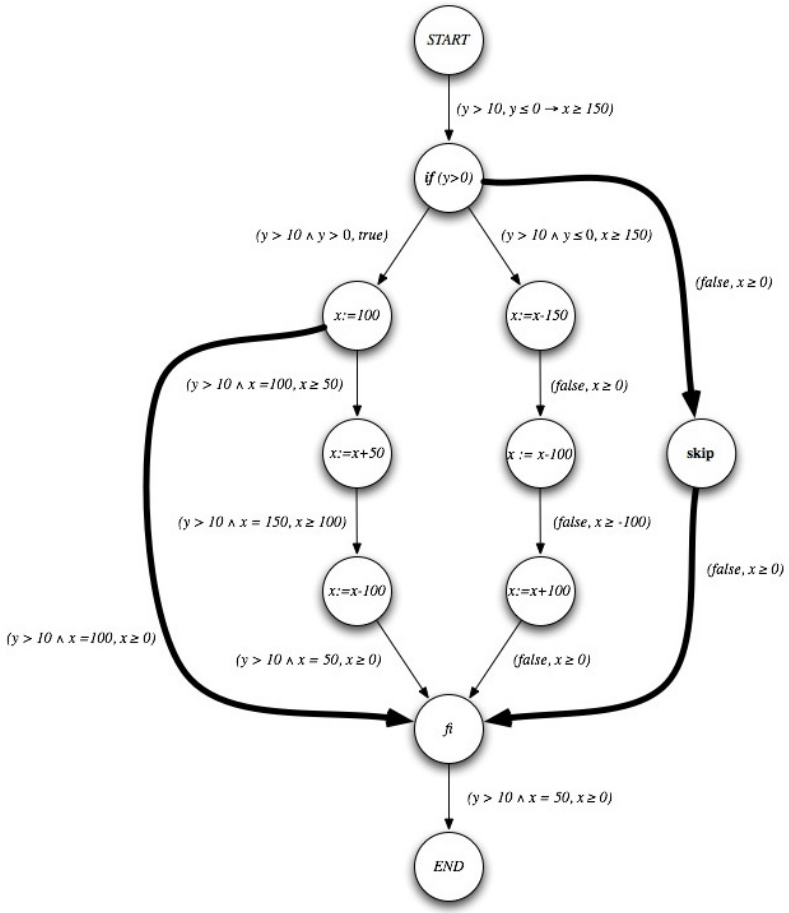

Figure 4. Example slice graph. Thick lines represent edges that were added to the initial CFG, corresponding to "shortcut" subprograms that do not modify the semantics of the program. These paths have the same source and destination nodes as other longer paths corresponding to removable sequences

Slicing Algorithm: Figure 5 shows the slice graphs for the two problematic examples presented in Section III. It is now quite clear that the notion of minimal slice *with respect to a given slice graph* is simply given by a read-back from the graph to the program. For each command sequence represented in the graph, we apply an (unweighted) shortest paths algorithm (basically a breadth-first traversal, linear on the size of the graph) to find a minimal slice of that sequence. Nodes that are not traversed correspond to commands that can be removed. This notion of minimality 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 conditions (and signaling them in the graph); the resulting slice will thus only be as good as the graph.

Slicing a loop involves slicing the loop's body recursively, with respect to the specification $(I, I \wedge \neg b)$). We remark however that the usefulness of this approach may be limited without human intervention. If the program is being sliced with a specification that has been weakened with respect to an initial, full specification, it makes sense to weaken the loop invariant accordingly, otherwise slicing the loop may result in no commands being removed at all inside its body.

VII. Conclusion

We are developing a laboratory[1] for experimenting with the ideas exposed here [12]. This is an online front-end for implementations of the algorithms reviewed and introduced in this paper. The user can choose between different precondition-based, postcondition-based, and specification-based slicing algorithms; the laboratory also offers standard verification capabilities (verification condition generation) and a visual representation of the LCF graphs introduced in the previous section. We find that many academic works in this area could and should have greater impact if prototype tools were offered to the community, and we are committed to developing such a tool, including the implementation of algorithms described in the literature in addition to our own.

[1]Available at `http://gamaepl.di.uminho.pt/gamaslicer`.

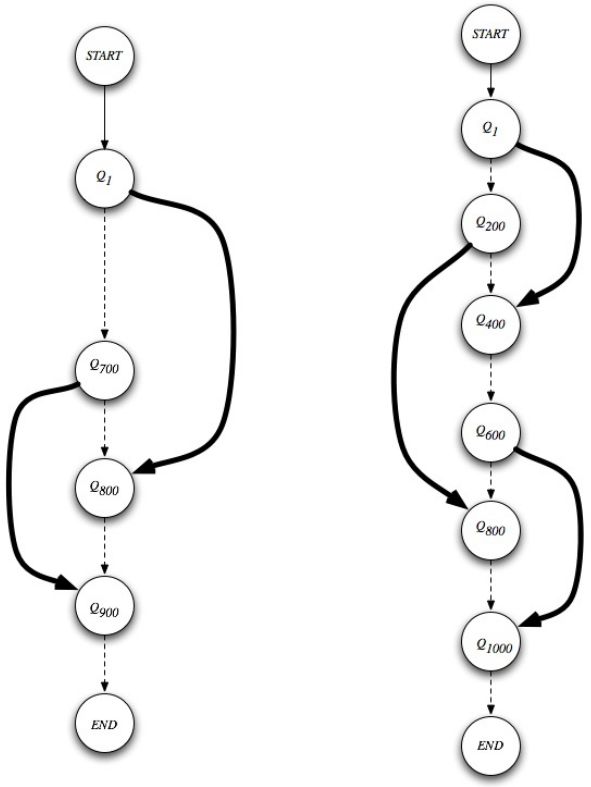

Figure 5. Example slice graphs

While the front-end is meant to allow for experimentation and comparison of different algorithms, we intend to optimise and test the graph-based algorithms with realistic code. One obligatory step will be to calculate weakest preconditions using Flanagan and Saxe's algorithm [11], which avoids the potential exponential explosion in the size of the conditions generated, keeping our algorithm within quadratic time.

As future work it will also be interesting to compare our approach with the work of Fox and colleagues [13], who introduced the *backward conditioning* technique, based on symbolic execution. The goal of this related approach is to remove from a program statements which, when executed, always lead to the negation of a given postcondition. The interest of this work is that it indicates that the interaction between slicing and verification happens in both directions: verification offers the tools used for implementing assertion-based slicing (of correct programs), but slicing can also be used to facilitate program verification.

Finally, we are convinced that the notion of control flow graph labeled with semantic information is of independent interest and may have other applications in program analysis, verification, and of course visualization.

Acknowledgment: This work was supported by project RESCUE, funded by FCT (PTDC/EIA/65862/2006).

References

[1] M. Weiser, "Program slicing," in *ICSE '81: Proceedings of the 5th international conference on Software engineering*. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.

[2] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[3] B. Meyer, "Applying "Design by Contract"," *IEEE Computer*, vol. 25, no. 10, 1992.

[4] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 3, pp. 212–232, 2005.

[5] M. Barnett, K. Rustan, M. Leino, and W. Schulte, "The Spec# programming system: An overview." in *CASSIS : construction and analysis of safe, secure, and interoperable smart devices*, vol. 3362. Springer, Berlin, March 2004, pp. 49–69.

[6] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, *ACSL: ANSI/ISO C Specification Language*, CEA and INRIA, preliminary design (version 1.4, October 29, 2008).

[7] G. Canfora, A. Cimitile, and A. D. Lucia, "Conditioned program slicing," *Information and Software Technology*, vol. 40, no. 11-12, pp. 595–608, November 1998, special issue on program slicing.

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

[9] J. J. Comuzzi and J. 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*. London, UK: Springer-Verlag, 1996, pp. 557–575.

[10] 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*. New York, NY, USA: ACM, 2001, pp. 605–609.

[11] C. Flanagan and J. 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*. New York, NY, USA: ACM, 2001, pp. 193–205.

[12] D. da Cruz, P. R. Henriques, and J. S. Pinto, "Gamaslicer: an Online Laboratory for Program Verification and Analysis," in *proceedings of the 10th. Workshop on Language Descriptions Tools and Applications (LDTA'10)*, 2010, to appear.

[13] C. Fox, S. Danicic, M. Harman, and R. M. Hierons, "Backward conditioning: A new program specialisation technique and its application to program comprehension," in *IWPC*. IEEE Computer Society, 2001, pp. 89–97.