ELSEVIER

Information and Software Technology 40 (1998) 595–607

INFORMATION AND SOFTWARE TECHNOLOGY

Conditioned program slicing

Gerardo Canfora*, Aniello Cimitile, Andrea De Lucia

Faculty of Engineering, University of Sannio, Palazzo Bosco Lucarelli, Piazza Roma, 82100 Benevento, Italy

Abstract

Slicing is a technique to decompose programs based on the analysis of the control and data flow. In the original Weiser's definition, a slice consists of any subset of program statements preserving the behaviour of the original program with respect to a program point and a subset of the program variables (slicing criterion), for any execution path. We present conditioned slicing, a general slicing model based on statement deletion. A conditioned slice consists of a subset of program statements which preserves the behaviour of the original program with respect to a slicing criterion for a given set of execution paths. The set of initial states of the program that characterise these paths is specified in the form of a first order logic formula on the input variables. We also show how slices deriving from other statement deletion based slicing models can be defined as conditioned slices. This is used to formally define a partial ordering relation between slicing models and to build a classification framework.

Keywords: Conditioned slicing; Program slicing

1. Introduction

Slicing is a technique to decompose programs based on the analysis of the control and data flow. In the original Weiser's definition [37], program slicing is based on the deletion of statements: a slice consists of any subset of program statements preserving the behaviour of the original program with respect to a program point and a subset of the program variables (*slicing criterion*). Although the computation of a minimal slice is an *undecidable* problem, an approximation can be computed by finding all the program statements that directly or indirectly affect the values of the variables in the subset at the selected program point. Program slicing has proven useful in several tasks, including testing [18,22], debugging [29,36], maintenance [17], complexity analysis [32], parallelisation [37], integration [23], comprehension [14], reverse engineering [31], re-engineering [28], and reuse [6,10,30]. A survey of program slicing techniques and their applications can be found in [34].

Weiser's slicing has also been called *static slicing*: a static slice preserves the behaviour of the original program with respect to the slicing criterion for any program execution. Several authors have demonstrated that a static slice can be too large to be analyzed and comprehended by a human reader [14,29]. To overcome this problem, alternative definitions of slicing have been proposed in the literature that restrict the concept of *preserving the program behaviour* to a set of program executions. Examples of alternative program slicing definitions are *dynamic slicing* [29], *quasi static slicing* [35], and *simultaneous dynamic slicing* [19], which compute slices with respect to a program input, a partial input, and a finite set of inputs, respectively.

In this paper we present conditioned slicing, a general slicing model based on statement deletion. A conditioned slice consists of a subset of program statements which preserves the behaviour of the original program with respect to a slicing criterion for any set of program executions. The set of initial states of the program that characterise these executions is specified in terms of a first order logic formula on the input. Conditioned slicing allows a better decomposition of the program giving human readers the possibility to analyze code fragments with respect to different perspectives. Conditioned slices can be computed by using symbolic execution and dependence graphs.

We also show how slices deriving from other statement deletion based slicing models can be defined as conditioned slices. This is used to formally define a partial ordering relation between slicing models, called *subsume relation*, and to build a classification framework.

The paper is organised as follows. Section 2 provides background information and recalls Weiser's definition of program slicing. Conditioned slicing is introduced in Section 3, which also discusses computation issues. Section 4 presents a framework for statement deletion based slicing models, including static, dynamic, simultaneous

*Corresponding author. E-mail: canfora@ingbn.unisa.it

0950-5849/98/$ – see front matter
PII: S0950-5849(98)00086-X

dynamic, quasi static, and conditioned slicing. The framework is based on the *subsume* relation, which forms a lattice on the slicing models. Section 5 gives some concluding remarks and discusses related work and software engineering applications of conditioned slicing.

2. Background and static slicing

This section provides background material and introduces program slicing as defined by Weiser [37]. Weiser's slicing is based on a program representation consisting of a graph, whose nodes represent program statements and whose edges represent transfers of the control.

Definition 2.1. A *digraph* is a tuple $G = (N,E)$, where N is a set of nodes and $E \subseteq N \times N$ is a set of edges. A *path* of length k from node n to node m is a sequence of nodes $\langle p_1,p_2,\ldots,p_k\rangle$ such that $p_1 = n$, $p_k = m$, and $\forall i$, $1 \le i \le k-1$, $(p_i,p_{i+1}) \in E$.

Definition 2.2. A *flowgraph* is a triple $FG = (N,E,n_0)$, where (N,E) is a digraph, $n_0 \in N$, and $\forall n \in N$ there is a path from n_0 to n.

Definition 2.3. A *hammock graph* is a quadruple $HG = (N,E,n_0,n_e)$, with the property that (N,E,n_0) and (N,E^{-1},n_e) are both flowgraphs, where $E^{-1} = \{(m,n)|(n,m) \in E\}$.

Any one-entry/one-exit program P is associated with a graph, called *control flow graph*, that describes its flow of control. A control flow graph is a hammock graph $HG = (N,E,n_0,n_e)$, where each node in N corresponds to a statement or a predicate. Statement nodes have one unlabelled outgoing edge, while predicate nodes have two outgoing edges, corresponding to conditional transfers of control, labelled *true* and *false*, respectively.

A program path from the entry node n_0 to the exit node n_e is *feasible* if there exist some input values which cause the path to be traversed during program execution.[1] A feasible path that has actually been executed for some input can be mapped onto the values the program variables assume before the execution of each statement. Such a mapping will be referred to as *state trajectory* [37]. An input to the program univocally determines a state trajectory.

Definition 2.4. A *state trajectory* of length k of a program P for input I is a finite sequence of ordered pairs $T = \langle(p_1,\sigma_1),(p_2,\sigma_2),\ldots,(p_k,\sigma_k)\rangle$, where $p_i \in N$, $1 \le i \le k$, $\langle p_1,p_2,\ldots,p_k\rangle$ is a path from n_0 to n_e, and σ_i, $1 \le i \le k$, is a function mapping the program variables onto the values they assume immediately before the execution of p_i.

[1] We assume program termination.

Weiser [37] defines a static program slice as any executable subset of program statements which preserves the behaviour of the original program at a program statement for a subset of program variables.

Definition 2.5. A *static slicing criterion* of a program P is a tuple $C = (p,V)$, where p is a statement in P and V is a subset of the variables in P.

A slicing criterion $C = (p,V)$ determines a projection function which selects from any state trajectory only the ordered pairs starting with p and restricts the variable-to-value mapping function σ to only the variables in V.

Definition 2.6. Let $C = (p,V)$ be a static slicing criterion of a program P and $T = \langle(p_1,\sigma_1),(p_2,\sigma_2),\ldots,(p_k,\sigma_k)\rangle$ be a state trajectory of P on input I. $\forall i$, $1 \le i \le k$:

$$\mathrm{Proj}'_C(p_i,\sigma_i) = \begin{cases} \lambda & \text{if } p_i \neq p, \\ \langle(p_i,\sigma_i|V)\rangle & \text{if } p_i = p, \end{cases}$$

where $\sigma_i|V$ denotes the restriction of the function σ_i to the domain V, and λ is the empty sequence. The extension of Proj'_C to the entire trajectory is defined as the concatenation of the result of the application of the function to the single pairs of the trajectory:

$$\mathrm{Proj}_C(T) = \mathrm{Proj}'_C(p_1,\sigma_1)\cdots\mathrm{Proj}'_C(p_k,\sigma_k).$$

A program slice is therefore defined behaviourally as any subset of program statements which preserves a specified projection of its behaviour.

Definition 2.7. A *static slice* of a program P on a static slicing criterion $C = (p,V)$ is any syntactically correct and executable program P' such that:

1. P' is obtained from P by deleting zero or more statements;
2. whenever P halts on input I with state trajectory T, then P' also halts on input I with state trajectory T' and $\mathrm{Proj}_C(T) = \mathrm{Proj}_C(T')$.

The above definition differs from the original definition of slice given in [37], because it requires that the instruction p always appears in the static slice.[2] This is not a limitation, considering that programmers can be easily confused if the instruction p of the slicing criterion is not included in the slice [29].

As an example, Fig. 1 shows a sample program which takes the integers `n`, `test0`, and a sequence of `n` integers `a` as input and computes the integers `possum`, `posprod`,

[2] Weiser's definition of slice requires that $\mathrm{Proj}_C(T) = \mathrm{Proj}_{C'}(T')$, where $C' = (\mathrm{succ}(p),V)$, and $\mathrm{succ}(p)$ is the nearest successor of p in the original program which is also in the slice, or p itself if p is in the slice.

```
1  main() {
2    int a, test0, n, i, posprod, negprod, possum, negsum, sum, prod;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = posprod = negprod = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7        scanf("%d", &a);
8        if  (a > 0) {
9             posssum += a;
10            posprod *= a; }
11       else if (a < 0) {
12                negsum -= a;
13                negprod *= (-a); }
14             else if (test0) {
15                      if (possum >= negsum)
16                           possum = 0;
17                      else negsum = 0;
18                      if (posprod >= negprod)
19                           posprod = 1;
20                      else negprod = 1; }
21       i++; }
22   if (possum >= negsum)
23        sum = possum;
24   else sum = negsum;
25   if (posprod >= negprod)
26        prod = posprod;
27   else prod = negprod; }
28   printf("%d \n", sum);
29   printf("%d \n", prod); }
```

Fig. 1. A sample program.

negsum, and negprod. The integers possum and negsum accumulate the sums of the positive numbers and of the absolute values of the negative numbers in the sequence, respectively. The integers posprod and negprod accumulate the products of the positive numbers and of the absolute values of the negative numbers in the sequence, respectively. Whenever an input a is zero, the greatest sum and the greatest product are reset if the value of test0 is not zero. The program returns the greatest sum and product computed. Fig. 2 shows a static slice of this program on the slicing criterion $C=(28, \texttt{sum})$.[3]

Whilst the problem of finding minimal static slices is *undecidable*, Weiser proposes an iterative algorithm based on data flow and on the *influence* of predicates on statement execution, which computes conservative slices guaranteed to have the properties of Definition 2.7. The slice is computed as the set of all statements of the program that might affect directly or indirectly the value of

[3]Where it is not ambiguous, we will refer to statements by using their line numbers.

```
1  main() {
2    int a, test0, n, i, possum, negsum, sum;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7        scanf("%d", &a);
8        if  (a > 0)
9             possum += a;
11       else if (a < 0)
12                negsum -= a;
14             else if (test0) {
15                      if (possum >= negsum)
16                           possum = 0;
17                      else negssum = 0;}
21       i++; }
22   if (possum >= negsum)
23        sum = possum;
24   else sum = negsum;
28   printf("%d \n", sum);
```

Fig. 2. A sample static slice.

the variables in V just before the execution of p. A parallel intraprocedural slicing algorithm has also been proposed by Danicic et al. [13].

Program slices can be computed using the *program dependence graph* [15,24] both at intraprocedural [33] and interprocedural level [24], and also in the presence of goto statements [8]. The program dependence graph is a program representation containing the same nodes as the control flow graph and two types of edges, *control dependence* edges and *data dependence* edges.

Usually, the data dependencies used in program slicing are flow dependencies corresponding to *def-use* triplets [2]. A def-use triplet (m,n,v) between two statements m and n, with respect to a variable v, arises if n uses the value of the variable v defined at m on some execution path. Although the problem of identifying def-use triplets is in general *undecidable*, an overly-conservative solution consists of considering triplets (m,n,v) where m defines the variable v, n uses v, and there is a path on the control flow graph from m to n on which v is not (re-)defined [2].

Control dependencies are defined as follows.

Definition 2.8. Let $HG = (N,E,n_0,n_e)$ be a control flow graph and n and m two nodes in N;

1. m *postdominates* n if every path from n to n_e contains m.
2. m is *control dependent* on n if:
   - there exists a path $\langle p_1,p_2,\ldots,p_k\rangle$, with $n = p_1$ and $m = p_k$, such that $\forall i,\ 2 \le i \le k-1$, m postdominates p_i;
   - m does not postdominate n.

Notice that if m is control dependent on n, then n has two outgoing edges (i.e., n corresponds to a predicate). Following one of the edges always results in m being executed, while taking the other edge may result in m not being executed. If the edge which always causes the execution of m is labelled with *true* (*false*, respectively), then m is control dependent on the *true* (*false*) branch of n.

A static slice can be computed by backward traversing control and data dependence edges on the dependence graph starting from the statement p of the slicing criterion and including in the slice all the statements and predicates reached by this transitive closure [33]. In this case the variables of the slicing criterion are the variables referenced in p.

3. A general program slicing model

Traditional static slicing considers subsets of the program statements with respect to all possible executions. However, there are cases when only a subset of the program executions are of interest. In this cases, static slices might be too large and include statements not involved in the executions of interest. Several different techniques have been introduced to compute program slices with respect to a subset of the program executions [14,19,20,29,35]. This is usually achieved by adding to the slicing criterion a specification of the set of initial states that trigger the desired executions. Each initial state is defined by a particular program input.

3.1. Conditioned slicing

In this section we describe a general program slicing model which allows any set of initial states of the program to be specified. This is done using a first order logic formula which maps a subset of the input program variables onto a set of initial states of the program. We call *conditioned slices* the slices resulting from adding such a condition to the slicing criterion.

Definition 3.1. Let V_{in} be a subset of input variables of a program P, and $F(V_{in})$ be a first order logic formula on the variables in V_{in}. A *conditioned slicing criterion* of a program P is a triple $C = (F(V_{in}),p,V)$, where p is a statement in P and V is a subset of the variables in P.

The first order logic formula F identifies a set of inputs to the program and consequently a set of state trajectories.

Definition 3.2. Let $V'_{in} = \{v_1,v_2,\ldots,v_n\}$ be the set of input variables of a program P. A *partial input* to the program for the variables in $V_{in} = \{v_1,v_2,\ldots,v_k\}$, $V_{in} \subseteq V'_{in}$, is any set $I = \{(v_1,i_1),(v_2,i_2),\ldots,(v_k,i_k)\}$, where i_i, $1 \le i \le k$, is a value of the same type of the variable v_i.

Definition 3.3. Let $V_{in} = \{v_1,v_2,\ldots,v_k\}$ be a set of input variables of a program P and $I = \{(v_1,i_1),(v_2,i_2,),\ldots,(v_k,i_k)\}$ be a partial input to the program. Let $V'_{in} = \{v_1,v_2,\ldots,v_k,\ldots,v_n\}$ be a set of input variables, such that $V_{in} \subseteq V'_{in}$. A *completion* of I with respect to V'_{in} is any partial input $I' = \{v_1,(i_1),(v_2,i_2),\ldots,(v_k,i_k),\ldots,(v_n,i_n)\}$. The set of all completions of I with respect to V' is denoted by $C(I,V')$.

Definition 3.4. Let V_{in} be a set of input variables of a program P and $F(V_{in})$ be a first order logic formula on the variables in V_{in}. A *satisfaction* for $F(V_{in})$ is any partial input I to the program for the variables in V_{in} that satisfies the formula F. The *satisfaction set* $S(F(V_{in}))$ is the set of all possible satisfactions for $F(V_{in})$.

If V_{in} is a subset of the input variables of the program P and $F(V_{in})$ is a first order logic formula on the variables in V_{in}, each completion $I' \in C(I,V'_{in})$, where $I \in S(F(V_{in}))$ and V'_{in} is the set of all input variables of the program P,

identifies a trajectory T.[4] A conditioned slice is any subset of the program statements that reproduces the original behaviour on each of these trajectories.

Definition 3.5. A *conditioned slice* of a program P on a conditioned slicing criterion $C = (F(V_{in}), p, V)$ is any syntactically correct and executable program P' such that:

1. P' is obtained from P by deleting zero or more statements;
2. whenever P halts on input I with state trajectory T, where $I \in C(I', V'_{in})$, $I' \in S(F(V_{in}))$, and V'_{in} is the set of input variables of P, then P' also halts on input I with state trajectory T' and $\mathrm{Proj}_{(p,V)}(T) = \mathrm{Proj}_{(p,V)}(T')$.

For example, Fig. 3 shows a conditioned slice of the program in Fig. 1 on the slicing criterion $C = (F(V_{in}), 28, \{\mathtt{sum}\})$, with $V_{in} = \{\mathtt{n}\} \cup_{1 \le i \le n} \{\mathtt{a}_i\}$ and $F(V_{in}) = (\forall i, 1 \le i \le \mathtt{n}, \mathtt{a}_i > 0)$. The condition F imposes that all the input values for the variable a be positive. This allows the program statements executed whenever this condition is not verified to be discarded from the conditioned slice.

3.2. Finding conditioned slices

A conditioned slice can be computed by first simplifying the program with respect to the condition on the input and then computing a slice on the reduced program. A symbolic executor [26,12] can be used to compute the reduced program, also called *conditioned program* in [6].

While in a traditional execution the values of program variables are constants, in symbolic execution they are represented by symbolic expressions, i.e. expressions containing symbolic constants. For example, the value v of a variable x might be represented by "$2*\alpha + \beta$", where α and β are symbolic constants. Moreover, unlike a program

```
1  main() {
2    int a, n, i, possum, negsum, sum;
3    scanf("%d", &n);
4    i = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7        scanf("%d", &a);
8        if (a > 0)
9             possum += a;
21       i++; }
22   if (possum >= negsum)
23        sum = possum;
28   printf("%d \n", sum);
```

Fig. 3. A sample conditioned slice.

[4] If a variable receives more than one value as input, then the different instances of such variable on the program trajectory are considered.

state in a traditional execution, a *symbolic state* is a pair (*State*, *PC*), where *State* is a set of pairs of the form (M, α), M and α being a memory location and its symbolic value, respectively, and *PC* is a first order logic formula, called *path-condition*, that identifies the control flow path traversed [26]. Indeed, while the evaluation of a predicate in a traditional execution unequivocally identifies the branch to follow, the symbolic evaluation of a predicate might generate two possible executions. For example, the execution of the predicate in line 8 of the program in Fig. 1, in the symbolic state:

$$(S_1, P_1) = \begin{array}{c}(\{(\mathtt{a}, \alpha_1), (\mathtt{test0}, \beta), (\mathtt{n}, \gamma), (\mathtt{i}, 1),\\ (\mathtt{posprod}, 1), (\mathtt{negprod}, 1),\\ (\mathtt{possum}, 0), (\mathtt{negsum}, 0),\\ (\mathtt{prod}, \mathit{undef}), (\mathtt{sum}, \mathit{undef}),\},\\ 1 \le \gamma),\end{array}$$

produces the two symbolic states:

$$(S_2, P_2) = \begin{array}{c}(\{(\mathtt{a}, \alpha_1), (\mathtt{test0}, \beta), (\mathtt{n}, \gamma), (\mathtt{i}, 1),\\ (\mathtt{posprod}, 1), (\mathtt{negprod}, 1),\\ (\mathtt{possum}, 0), (\mathtt{negsum}, 0),\\ (\mathtt{prod}, \mathit{undef}), (\mathtt{sum}, \mathit{undef}),\},\\ 1 \le \gamma \wedge \alpha_1 > 0),\end{array}$$

$$(S_3, P_3) = \begin{array}{c}(\{(\mathtt{a}, \alpha_1), (\mathtt{test0}, \beta), (\mathtt{n}, \gamma), (\mathtt{i}, 1),\\ (\mathtt{posprod}, 1), (\mathtt{negprod}, 1),\\ (\mathtt{possum}, 0), (\mathtt{negsum}, 0),\\ (\mathtt{prod}, \mathit{undef}), (\mathtt{sum}, \mathit{undef}),\},\\ 1 \le \gamma \wedge \alpha_1 \le 0).\end{array}$$

The two path-conditions carry the conditions that must be satisfied in order to follow the true or false branch, respectively. The path-condition can be used to discard infeasible paths. This is particularly true whenever the symbolic execution of a program is made with an initial non-trivial path-condition. For example, let us symbolically execute the program in Fig. 1 with the initial path-condition[5] $\forall i, 1 \le i \le \gamma, \alpha_i > 0$, where γ is the input symbolic value for n and α_i is the input symbolic value for $\mathtt{a}_i$, $1 \le i \le \gamma$. The symbolic state before the execution of the predicate in line 8 will be:

$$(S'_1, P'_1) = \begin{array}{c}(\{(\mathtt{a}, \alpha_1), (\mathtt{test0}, \beta), (\mathtt{n}, \gamma), (\mathtt{i}, 1),\\ (\mathtt{posprod}, 1), (\mathtt{negprod}, 1),\\ (\mathtt{possum}, 0), (\mathtt{negsum}, 0),\\ (\mathtt{prod}\ \mathit{undef}), (\mathtt{sum}, \mathit{undef}),\},\\ (\forall i, 1 \le i \le \gamma, \alpha_i > 0) \wedge 1 \le \gamma).\end{array}$$

Due to the initial condition, the path-condition P'_1 implies the condition $\alpha_1 > 0$ deriving from the evaluation of the predicate in line 8 in the state S'_1. Therefore, the *false*

[5] The initial path-condition corresponds to the condition ($\forall i, 1 \le i \le \mathtt{n}, \mathtt{a}_i > 0$) of the slicing criterion that defines the conditioned slice in Fig. 3.

branch can be discarded and the execution will continue on the *true* branch with the symbolic state unchanged.

The evaluation of such implications is in general an *undecidable* problem. However, in most cases the symbolic expressions can be simplified and these implications can be automatically evaluated by a theorem prover, e.g. [4]. Fig. 4 shows the conditioned program resulting from the symbolic execution of the program in Fig. 1, with the given initial path-condition. Notice that whenever the condition $(\forall i, 1 \leq i \leq \mathtt{n}, \mathtt{a}_i > 0)$ holds true, the predicates `possum >= negsum` and `posprod >= negprod` also hold true and then their *false* branches are not considered for inclusion in the conditioned program.

Problems can arise in symbolic execution of loops whenever the current path-condition does not imply either the loop condition or its negation. A solution for this problem requires in general the determination of a suitable *loop invariant* which allows symbolic execution to continue at the end of the loop (see [9] for a survey of such techniques). However, for the purpose of conditioned slicing, we only need to know which statements can be executed within the loop, in order to construct the conditioned program. In this case, the symbolic execution of a loop can be driven by humans that can decide the number of times the loop must be executed: whenever the path to be followed is chosen, the path-condition is modified accordingly. For example, for the *while* loop of the program in Fig. 1, one loop execution is needed for discarding infeasible paths with respect to the initial condition. In some cases, the user can also try to generalise the sample executions and recover a loop invariant for the sake of precision.

Intuitively, a conditioned slice could also be computed by first computing a static slice and then simplifying it through symbolic execution. However, this is equivalent to intersecting the static slice and the conditioned program, which has been proven to produce larger slices than slicing the conditioned program [6].

Dependence graphs can be used to compute conditioned slices from conditioned programs. A first simple solution consists of marking all the nodes of a dependence graph that correspond to statements and predicates symbolically executed with the initial path-condition. Then a conditioned slice can be computed by backward traversing control and data dependence edges between marked nodes and including in the slice all the statements and predicates reached by this transitive closure [6].

However, this solution only considers static dependencies and might produce overly-conservative slices. As an example, let us consider the following code fragment:

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

From a static point of view, the data dependencies (1, 4, x) and (3, 4, x) are represented on a program dependence graph (see Fig. 5). Therefore, a static slice on the slicing criterion (4, {z}) will include the entire code fragment. On the contrary, a conditioned slice on the slicing criterion (C, 4, {z}), where the condition C is such that $C \Rightarrow \mathtt{a} > 0$, should not include the statement in line 1. Indeed, whenever the condition C holds true, the statement in line 3 is always executed and then no path generating the dependency (1, 4, x) is executed. However, using static dependencies between marked nodes of the dependence graph would include the statement in line 1 in the conditioned slice.

```
1  main() {
2    int a, test0, n, i, posprod, negprod, possum, negsum, sum, prod;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = posprod = negprod = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7        scanf("%d", &a);
8        if  (a > 0) {
9             possum += a;
10            posprod *= a; }
21       i++; }
22   if (possum >= negsum)
23        sum = possum;
25   if (posprod >= negprod)
26        prod = posprod;
28   printf("%d \n", sum);
29   printf("%d \n", prod); }
```

Fig. 4. A sample conditioned program.

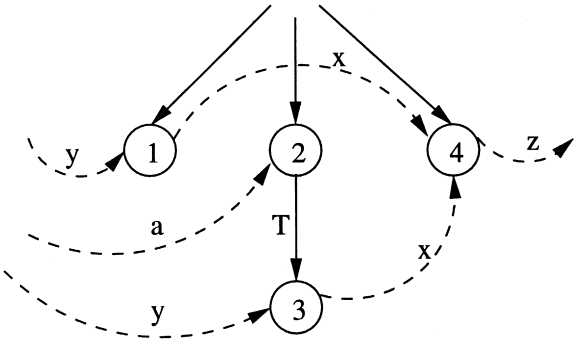

Fig. 5. A program dependence graph fragment.

To overcome this problem, a different approach consists of marking the data dependencies for which there exists a path traversed during symbolic execution. The conditioned slice can be obtained by only considering the marked nodes and edges during the backward traversal of the dependence graph. For example, the symbolic execution of the code fragment above will mark all the statements and the data dependency (3, 4, x), but it will not mark the data dependency (1, 4, x). This allows us to discard the statement in line 1 as it cannot be reached on marked edges. Fig. 6 shows the conditioned slicing algorithm. The procedure *ConditionedSlice* takes a *marked* program dependence graph (PDG) and a node p corresponding to the statement of the slicing criterion as its input and returns the set of nodes corresponding to statements to be included in the conditioned slice. Of course, if the node p is not marked, i.e. the corresponding statement is not included in the conditioned program, the resulting conditioned slice is empty. Otherwise, the slice will include all nodes reachable on marked edges during the backward traversal of the dependence graph; $m \rightarrow_{cd} n$ and $m \rightarrow_{du} n$ denote control and data dependence edges, respectively.

```
procedure ConditionedSlice(G, p, Slicelist)
    declare
        G: a marked PDG;
        p, n, m: PDG nodes;
        Slicelist, Worklist: sets of PDG nodes;
    begin
        Slicelist ← ∅;
        if marked(p) then
            Worklist ← {p};
            while Worklist ≠ ∅ do
                Select and remove a node n from Worklist;
                Slicelist ← Slicelist ∪ {n};
                for all m such that m ∉ Slicelist and
                        (m →x n with x ∈ {cd, du}) and
                        marked(m →x n)
                    Worklist ← Worklist ∪ {m};
                endfor
            endwhile
        endif
    end
```

Fig. 6. The conditioned slicing algorithm.

4. A program slicing framework

In this section we discuss some of the other statement deletion based program slicing models presented in the literature and show that conditioned slicing is a general slicing model which subsumes all the models discussed.

4.1. Dynamic slicing

Program slicing was first proposed as a tool for decomposing programs during debugging, in order to allow a better understanding of the portion of code which revealed an error [36,37]. In this case the slicing criterion contains the variables which produced an unexpected result on some input to the program. However, a static slice may very often contain statements which have no influence on the values of the variables of interest for the particular execution in which the anomalous behaviour of the program was discovered.

Korel and Lasky [29] propose a refinement of static slicing, called *dynamic slicing*, which uses dynamic analysis to identify all and only the statements that affect the variables of interest on the particular anomalous execution. In this way, the size of the slice can be considerably reduced, thus allowing an easier localisation of the bugs. Another advantage of dynamic slicing is the run-time handling of arrays and pointer variables. Dynamic slicing treats each element of an array individually, whereas static slicing considers each definition or use of any array element as a definition or use of the entire array. Similarly, dynamic slicing distinguishes which objects are pointed to by pointer variables during a program execution.

From a formal point of view, a dynamic slice is defined with respect to a particular trajectory: the slicing criterion refers to a statement in any particular position in the state trajectory.[6] In this paper, we assume that a slicing criterion always refers to the last occurrence of a statement in a trajectory. As a consequence, the only difference between static and dynamic slicing is that a dynamic slice is required to preserve the behaviour of the original program on only one input, whereas the static slice must be correct on any input. In Ref. [19] the same assumption is implicitly made, while Agrawal and Horgan [1] compute

[6] In the original definition [29], a *dynamic slicing criterion* of a program P executed on input I is a triple $C = (I,i,V)$, where $1 \le i \le k$ is a position in the trajectory $T = \langle (p_1,\sigma_1),(p_2,\sigma_2),\ldots,(p_k,\sigma_k)\rangle$ obtained by executing P on input I, and V is a subset of the variables in P. A *dynamic slice* of P on C is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and when executed on input I produces a trajectory $T' = \langle (p'_1,\sigma'_1),(p'_2,\sigma'_2),\ldots,(p'_{k'},\sigma'_{k'})\rangle$ for which there exists an execution position i', $1 \le i' \le k'$, such that:

1. $\langle p'_1,p'_2,\ldots,p'_{i'}\rangle$ is obtained by removing from $\langle p_1,p_2,\ldots,p_i\rangle$ all the occurrences of statements that are not in P';
2. $\mathrm{Proj}_{(p_i,V)}(p_i,\sigma_i) = \mathrm{Proj}_{(p_{i'},V)}(p'_{i'},\sigma'_{i'})$.

dynamic slices with respect to the last statement of the program.

Definition 4.1. A *dynamic slicing criterion* of a program P executed on input I is a triple $C = (I, p, V)$, where p is a statement in P and V is a subset of the variables in P.

Definition 4.2. A *dynamic slice* of a program P on a dynamic slicing criterion $C = (I, p, V)$ is any syntactically correct and executable program P' such that:

1. P' is obtained from P by deleting zero or more statements;
2. whenever P halts on input I with state trajectory T, then P' also halts on input I with state trajectory T' and $\mathrm{Proj}_{(p,V)}(T) = \mathrm{Proj}_{(p,V)}(T')$.

For example, Fig. 7 shows a dynamic slice of the program in Fig. 1 on the slicing criterion $C = (I, 28, \{\texttt{sum}\})$, where $I = \{(\texttt{test0}, 0), (\texttt{n}, 2), (\texttt{a}_1, 0), (\texttt{a}_2, 2)\}$.[7]

Korel and Lasky [29] propose an iterative algorithm to compute dynamic slices. The algorithm requires that if any occurrence of a statement (within a loop) in the trajectory is included in the slice, then all the other occurrences of that statement be included in the slice. This ensures that the slice extracted is executable. Other algorithms use dynamic dependence graphs [1,25] to produce more refined slices. They consider only the occurrences of statements in the trajectory that affect the computation of the variables in the slicing criterion. The resulting slices are not necessarily an executable subset of the original program.

```
1  main() {
2    int a, test0, n, i, possum, negsum, sum;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7       scanf("%d", &a);
8       if  (a > 0)
9             possum += a;
21      i++; }
22   if (possum >= negsum)
23       sum = posum;
28   printf("%d \n", sum);
```

Fig. 7. A sample dynamic slice.

[7]The subscripts refer to the different occurrences of the input variable a within the different loop iterations.

4.2. Quasi static slicing

Venkatesh [35] proposes a notion of slicing that falls between static and dynamic slicing. The need for such a slice, called *quasi static* slice, arises from applications where the value of some input variables is fixed while the behaviour of the program must be analyzed when other input values vary.

Definition 4.3. Let V_{in} be a subset of input variables of a program P and I be a partial input to P for the variables in V_{in}. A *quasi static slicing criterion* of a program P is a triple $C = (I, p, V)$, where p is a statement in P and V is a subset of the variables in P.

If V'_{in} is the set of input variables of the program P, each completion $I' \in C(I, V'_{\mathrm{in}})$, for some partial input I, identifies a trajectory T. A quasi static slice is any subset of the program which reproduces the original behaviour on each of these trajectories.

Definition 4.4. A *quasi static slice* of a program P on a quasi static slicing criterion $C = (I, p, V)$ is any syntactically correct and executable program P' such that:

1. P' is obtained from P by deleting zero or more statements;
2. whenever P halts on input I', where $I' \in C(I, V'_{\mathrm{in}})$ and V'_{in} is the set of input variables of P, with state trajectory T, then P' also halts on input I' with state trajectory T' and $\mathrm{Proj}_{(p,V)}(T) = \mathrm{Proj}_{(p,V)}(T')$.

For example, Fig. 8 shows a quasi static slice of the program in Fig. 1 on the slicing criterion $C = (\{(\texttt{test0}, 0)\}, 28, \{\texttt{sum}\})$.

```
1  main() {
2    int a, test0, n, i, possum, negsum, sum;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7       scanf("%d", &a);
8       if  (a > 0)
9            possum += a;
11      else if (a < 0)
12               negsum -= a;
21      i++; }
22   if (possum >= negsum)
23       sum = possum;
24   else sum = negsum;
28   printf("%d \n", sum);
```

Fig. 8. A sample quasi static slice.

A quasi static slice is constructed with respect to an initial prefix I' of the input sequence. This is closely related to partial evaluation or mixed computation [3], a technique to specialise programs with respect to partial inputs. By specifying the values of some of the input variables, constant propagation and simplification can be used to reduce expressions to constants. In this way, the values of some program predicates can be evaluated, thus allowing the deletion of branches which are not executed on the particular partial input. Quasi static slices are computed on specialised programs.

4.3. Simultaneous dynamic slicing

A different model to compute slices with respect to a set of executions of the program has been proposed by Hall [19]. The model is called *simultaneous dynamic program slicing* because it extends dynamic slicing and simultaneously applies it to a set of test cases.

Definition 4.5. Let $\{T_1,T_2,\ldots,T_m\}$ be a set of state trajectories of length $k_1,k_2,\ldots,k_m$, respectively, of a program P on input $\{I_1,I_2,\ldots,I_m\}$. A *simultaneous dynamic slicing criterion* of P executed on each of the input I_j, $1 \le j \le m$, is a triple $C = (\{I_1,I_2,\ldots,I_m\},p,V)$, where p is a statement in P, and V is a subset of the variables in P.

Definition 4.6. A *simultaneous dynamic slice* of a program P on a simultaneous dynamic slicing criterion $C = (\{I_1,I_2,\ldots,I_m\},p,V)$ is any syntactically correct and executable program P' such that:

1. P' is obtained from P by deleting zero or more statements;
2. whenever P halts on input I_j, $1 \le j \le m$, with state trajectory T_j, then P' also halts on input I_j with state trajectory T'_j and $\mathrm{Proj}_{(p,V)}(T_j) = \mathrm{Proj}_{(p,V)}(T'_j)$.

For example, Fig. 9 shows a simultaneous dynamic slice of the program in Fig. 1 on the slicing criterion $C = (\{I_1,I_2\}, 28, \{\texttt{sum}\})$, where $I_1 = \{(\texttt{test0}, 0), (\texttt{n}, 2), (\texttt{a}_1, 0), (\texttt{a}_2, 2)\}$ and $I_2 = \{(\texttt{test0}, 1), (\texttt{n}, 2), (\texttt{a}_1, 0), (\texttt{a}_2, 2)\}$.

A simultaneous program slice on a set of test cases is not simply given by the union of the dynamic slices on the component test cases. Indeed, simply unioning dynamic slices is unsound, in that the union does not maintain simultaneous correctness on all the inputs. Ref. [19] presents an iterative algorithm that, starting from an initial set of statements, incrementally constructs the simultaneous dynamic slice, by computing at each iteration a larger dynamic slice.

```
1  main() {
2    int a, test0, n, i, possum, negsum, sum;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7        scanf("%d", &a);
8        if  (a > 0)
9             possum += a;
11       else if (a < 0) {}
14            else if (test0)
15                      if (possum >= negsum)
16                           possum = 0;
21     i++; }
22   if (possum >= negsum)
23         sum = possum;
28   printf("%d \n", sum);
```

Fig. 9. A sample simultaneous dynamic slice.

4.4. Relationships between program slicing models

In this section we present a framework for statement deletion based program slicing. The slicing models discussed in the previous sections can be classified according to a partial ordering relation, called *subsume relation*, based on the sets of program inputs specified by the slicing criteria. Indeed, for each of these slicing models, a slice preserves the behaviour of the original program on all the trajectories identified by the set of program inputs specified by the slicing criterion.

Definition 4.7. A program slicing model SM_1 *subsumes* a program slicing model SM_2 if for each slicing criterion defined according to SM_2 there exists an equivalent slicing criterion defined according to SM_1 that specifies the same set of program inputs.

A slicing model is stronger than the slicing models it subsumes, because it is able to specify and compute slices with respect to a broader set of slicing criteria. Consequently, any slice computed according to a slicing model can also be computed with a stronger slicing model. The subsume relation defines a hierarchy on the statement deletion based program slicing models described in this paper and the root of this hierarchy is conditioned slicing. This means that conditioned slicing subsumes any other slicing model. Fig. 10 shows the *subsume* hierarchy.

Theorem 4.1. *The set of slicing models {static slicing, dynamic slicing, quasi static slicing, simultaneous dynamic slicing, conditioned slicing} is partially ordered with respect to the subsume relation. The following relationships hold:*

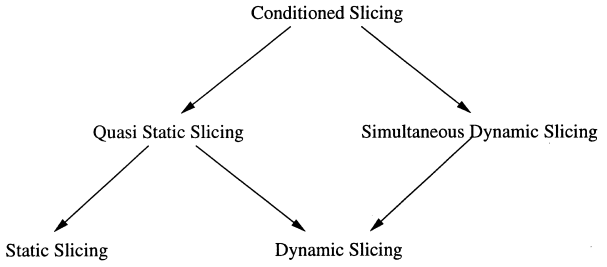

Fig. 10. Relationships between program slicing models.

1. *Quasi static slicing subsumes static slicing.*
2. *Quasi static slicing subsumes dynamic slicing.*
3. *Simultaneous dynamic slicing subsumes dynamic slicing.*
4. *Conditioned slicing subsumes quasi static slicing.*
5. *Conditioned slicing subsumes simultaneous dynamic slicing.*
6. *There is no relation between static slicing and dynamic slicing and between quasi static slicing and simultaneous dynamic slicing.*

Proof. Let P be a program, p be a program statement, V be a set of program variables, and $V'_{\text{in}} = \{v_1, v_2, \ldots, v_n\}$ be the set of input variables of P.

1. *Quasi static slicing subsumes static slicing.* Let $C = (p,V)$ be a static slicing criterion. The set of completions $C(I,V'_{\text{in}})$ corresponds to the set of all possible inputs to the program, whenever $I = \varnothing$. Therefore, the quasi static slicing criterion $(\varnothing,p,V)$ specifies the same set of program inputs as C.
2. *Quasi static slicing subsumes dynamic slicing.* Let I be a program input and $C = (I,p,V)$ be a dynamic slicing criterion. Of course, I is a partial input to the program and therefore C is also a quasi static slicing criterion.
3. *Simultaneous dynamic slicing subsumes dynamic slicing.* Let I_1 be a program input and $C = (I_1,p,V)$ be a dynamic slicing criterion. Of course, C is also a simultaneous dynamic slicing criterion $(\{I_1, I_2, \ldots, I_m\},p,V)$, where $m = 1$.
4. *Conditioned slicing subsumes quasi static slicing.* Let $V_{\text{in}} = \{v_1, v_2, \ldots, v_k\}$, $1 \le k \le n$, be a subset of V'_{in}, $I = \{(v_1,i_1),(v_2,i_2), \ldots, (v_k,i_k)\}$ be a partial input, and $C = (I,p,V)$ be a quasi static slicing criterion. The satisfaction set $S(F(V_{\text{in}}))$ of the first order logic formula[8] $F(V_{\text{in}}) = (v_1 = i_1 \wedge v_2 = i_2 \wedge \ldots \wedge v_k = i_k)$ coincides with $\{I\}$. Therefore, the conditioned slicing criterion $(F(V_{\text{in}}),p,V)$ and the quasi static slicing criterion C are equivalent, as they specify the same set of program inputs.
5. *Conditioned slicing subsumes simultaneous dynamic slicing.* Let $\{I_1, I_2, \ldots, I_m\}$ be a set of program inputs, where $I_j = \{(v_1,i_{j,1}),(v_2,i_{j,2}), \ldots, (v_n,i_{j,n})\}$, $1 \le j \le m$, and $C = (\{I_1, I_2, \ldots, I_m\},p,V)$ be a simultaneous dynamic slicing criterion. The satisfaction set of the first order logic formula $S(F(V'_{\text{in}})) = ((v_1 = i_{1,1} \wedge v_2 = i_{1,2} \wedge \ldots \wedge v_n = i_{1,n}) \vee ((v_1 = i_{2,1} \wedge v_2 = i_{2,2} \wedge \ldots \wedge v_n = i_{2,n}) \vee \ldots \vee ((v_1 = i_{m,1} \wedge v_2 = i_{m,2} \wedge \ldots \wedge v_n = i_{m,n}))$ coincides with $\{I_1, I_2, \ldots, I_m\}$. Moreover, for each $I \in S(F(V'_{\text{in}}))$ the completion set $C(I,V'_{\text{in}})$ coincides with I. Therefore, the conditioned slicing criterion $(F(V_{\text{in}}),p,V)$ and the simultaneous slicing criterion C are equivalent, as they specify the same set of program inputs.
6. *There is no relation between static slicing and dynamic slicing and between quasi static slicing and simultaneous dynamic slicing.* This statement derives directly from the fact that, while static slicing and quasi static slicing criteria can specify infinite sets of program inputs, dynamic slicing and simultaneous dynamic slicing criteria can only specify finite sets of program inputs. □

The theorem states that any slicing criterion can be expressed as a conditioned slicing criterion, i.e. any slice can be computed by suitably specifying a set of variables, a program point and a condition on the program input. For example, with reference to the program shown in Fig. 1, the static slicing criterion $(28, \{\texttt{sum}\})$ can be expressed as $(\textit{true}, 28, \{\texttt{sum}\})$ and the dynamic slicing criterion $(I, 28, \{\texttt{sum}\})$, where $I = \{(\texttt{test0}, 0), (\texttt{n}, 2), (\texttt{a}_1, 0), (\texttt{a}_2, 2)\}$, can be expressed as $(F(V_{\text{in}}), 28, \{\texttt{sum}\})$, where $V_{\text{in}} = \{\texttt{test0}, \texttt{n}, \texttt{a}_1, \texttt{a}_2\}$ and $F(V_{\text{in}}) = (\texttt{test0} = 0 \wedge \texttt{n} = 2 \wedge \texttt{a}_1 = 0 \wedge \texttt{a}_2 = 2)$. The quasi static criterion $(\{(\texttt{test0}, 0)\}, 28, \{\texttt{sum}\})$ can be expressed as $(\texttt{test0} = 0, 28, \{\texttt{sum}\})$ and the simultaneous dynamic slicing criterion $(\{I_1, I_2\}, 28, \{\texttt{sum}\})$, where $I_1 = \{(\texttt{test0}, 0), (\texttt{n}, 2), (\texttt{a}_1, 0), (\texttt{a}_2, 2)\}$ and $I_2 = \{(\texttt{test0}, 1), (\texttt{n}, 2), (\texttt{a}_1, 0), (\texttt{a}_2, 2)\}$, is $(F(V_{\text{in}}), 28, \{\texttt{sum}\})$, where $V_{\text{in}} = \{\texttt{test0}, \texttt{n}, \texttt{a}_1, \texttt{a}_2\}$ and $F(V_{\text{in}}) = ((\texttt{test0} = 0 \wedge \texttt{n} = 2 \wedge \texttt{a}_1 = 0 \wedge \texttt{a}_2 = 2) \vee (\texttt{test0} = 1 \wedge \texttt{n} = 2 \wedge \texttt{a}_1 = 0 \wedge \texttt{a}_2 = 2))$.

5. Concluding remarks

In this paper we have presented conditioned slicing and a general framework for statement deletion based program slicing models, including static slicing [37], dynamic slicing [29], quasi static slicing [35], and simultaneous dynamic slicing [19]. Each model is able to specify program slices that preserve the behaviour of the original program on the set of program executions specified by the slicing criterion. The traditional static and dynamic slicing models consider slices with respect to either all possible executions or just one execution, respectively. Simultaneous dynamic slicing and quasi static slicing can specify

[8] The symbols $\wedge$ and $\vee$ denote the logical connectors *and* and *or*, respectively.

slices with respect to a set of program inputs by explicitly using a finite set of test cases or by assigning a value to only a subset of the input variables, respectively. Conditioned slicing is the most general model as it enables the specification of slices with respect to any set of program inputs. This is achieved by adding a first order logic formula on the input variables to the slicing criterion.

Conditioned slices can be computed by first simplifying the program with respect to the condition on the input, and then computing the slices on the conditioned program. A symbolic executor can be used to compute the conditioned program, while program dependencies are exploited for isolating the program slice. We have implemented a conditioned slicer for C programs in Prolog. A fine-grained representation for C programs, called the Combined C Graph (CCG) [27] (consisting of an abstract syntax tree combined with a control flow graph and program dependencies) is used both to simplify the program and to compute the slices [9,10]. Future work will be in the direction of extending conditioned slicing to the interprocedural level.

5.1. Related work

There are a number of other related approaches that use different definitions of slicing to compute subsets of program statements that exhibit a particular program behaviour. All these approaches add information to the slicing criterion to reduce the size of the computed slices. For example, Lanubile and Visaggio [30] introduce the concept of *transform slicing* to isolate reusable functions. A transform slicing criterion contains an additional set of variables used to stop the computation of the slice as soon as the definitions of these variables are reached during the backward traversal of the control flow graph. Cimitile et al. [10] add a second statement to the slicing criterion and compute slices comprised between the two statements. These approaches are based on static slicing; conditioned slicing can be extended to include in the slicing criterion the additional set of variables or the additional statement.

Coen-Porisini et al. [11] use symbolic execution to specialise generalised software components to more specific and efficient functions to be used under more restricted conditions. The specialisation is accomplished by symbolically executing the source module with an initial path-condition, i.e. a first order logic formula which restricts the application domain of the generalised component. Conditioned slicing produces more specialised functions by also restricting the set of output variables.

Approaches more related to conditioned slicing have been presented by Ning et al. [31] and Field et al. [16]. Ning et al. [31] introduce a tool, called COBOL/SRE, to extract different types of slices from legacy systems, in particular *condition-based* slices. The user specifies a logical expression and a slicing range and the tool automatically isolates the statements that can be reached along control flow paths under the given condition. The authors do not give a formal definition of condition-based slice. Field et al. [16] introduce the concept of *constrained* slice to indicate slices that can be computed with respect to any set of constraints. Their approach is based on an intermediate representation for imperative programs, named PIM, and exploits graph rewriting techniques based on *dynamic dependence tracking* that model symbolic execution. The slices extracted are not executable. The authors are interested in the semantic aspects of more complex program transformations rather than in simple statement deletion.

The framework of statement deletion based slicing models presented in this paper can be extended with more powerful simplification rules than statement deletion. Harman and Danicic [21] introduce *amorphous program slicing*. Like a traditional slice, an amorphous slice preserves a projection of the semantics of the original program from which it is constructed. However, it can be computed by applying a broader range of transformation rules, including statement deletion. This is particularly useful in program comprehension, as more powerful transformations may sensibly simplify complex programs.

5.2. Applications of conditioned slicing

There are several software engineering applications of conditioned program slicing. Any application where static slicing produces too generic components and dynamic slicing produces too specific components would benefit from using conditioned slicing. In this section we discuss conditioned slicing in the context of program comprehension and reuse of existing software.

A widely accepted theory to explain the way programmers comprehend code is the Brooks's [5] *top-down* comprehension model. This theory views comprehension as the process of recovering the decisions that a program designer makes when designing a program. The recovery of design decisions progresses through an iterative process of guessing, formulating *hypotheses*, and verifying them. Hypotheses must be verified, i.e. confronted with the evidence found in the code. During this verification the hypotheses are either confirmed or discarded. The verification of hypotheses is most commonly done against static code, i.e. by scanning code for *beacons*. A beacon is a code fragment that matches the current hypothesis. Hypotheses are iteratively refined, producing new sub-goals to be verified by scanning code again.

A technique that programmers may use when scanning code for beacons is program slicing. The traditional static slicing model restricts the type of hypotheses that can be specified to those concerning the output of an expected function and the place where this is computed. However, a function can behave differently depending on particular conditions. For example, the price for renting a car and the insurance rate might depend on the type of car, the age of

the driver, and the duration of the renting period. Whenever a maintainer has knowledge of the application domain, he/she would like to recover the different function behaviours by isolating the portion of code corresponding to each of them. Conditioned slicing enables the computation of refined code fragments implementing specific function behaviours. Hypotheses about the conditions that characterise each behaviour can be formulated and expressed in the form of a first order logic formula on a subset of the program input variables.

Using existing software to build reusable parts has emerged as a key to spread the practice of reuse. Many code scavenging methods have been proposed in the literature that exploit different technologies and tools (see [7] for an overview). Most of these methods search existing systems for software components that satisfy particular structural properties: these methods are called *structural* or *mass* methods, as they produce a large set of candidates for reuse. Alternatively, reusable parts can be searched based on the specification of a desired function.

Program slicing can be used to isolate a function within a larger program. A static slice implementing a function can be computed once the specification of the function is mapped onto a slicing criterion [6,30]. In particular, the output of the function can be mapped onto the variables of the criterion and the specification post-condition can be mapped onto the statement of the criterion [10]. Conditioned slicing enables the isolation of more refined code fragments whenever the reusable function has to be specialised with respect to a particular behaviour. The latter is specified through the condition of the slicing criterion.

Conditioned slicing has been applied in two controlled experiments together with other specification driven slicing techniques for the comprehension and the isolation of code fragments implementing reusable functions [6,10]. The first case study consisted of a COBOL software system in the banking domain. We applied conditioned slicing to a subsystem sized about 2000 LOC, for the management of current accounts and pass-books: 46 slices were isolated that implemented meaningful functions, such as depositing and drawing, verifying/modifying the data of an account's owner, exchanging cheques, opening and closing an account, managing a transaction, and printing the statement of a current account. The second case study consisted of a medium sized C software system implementing the Combined C Graph static analyser [27]. We applied conditioned slicing to the four largest functions (about 1000 LOC total) implementing the output functionality of the analyser. As a result of the experiment we decomposed the original functions into 18 smaller functions, each of which was sized less than 150 LOC.

References

[1] H. Agrawal, J.R. Horgan, Dynamic program slicing, in: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, White Plains, New York, ACM Press, 1990, pp. 246–256.

[2] A.V. Aho, R. Sethi, J.D. Ullmann, Compilers: Principles, Techniques and Tools, Addison-Wesley, Reading, MA, 1986.

[3] D. Bjørner, A.P. Ershov, N.D. Jones, Partial Evaluation and Mixed Computation, North-Holland, Amsterdam, 1987.

[4] R.S. Boyer, J.S. Moore, A Computational Logic, Academic Press, New York, 1979.

[5] R. Brooks, Towards a theory of the comprehension of computer programs, International Journal of Man–Machines Studies 18(6) (1983) 543–554.

[6] G. Canfora, A. Cimitile, A. De Lucia, G.A. Di Lucca, Software salvaging based on conditions, in: Proceedings of International Conference on Software Maintenance, Victoria, British Columbia, Canada, IEEE CS Press, 1994, pp. 424–433.

[7] G. Canfora, A. Cimitile, G. Visaggio, Assessing modularisation and code scavenging techniques, Journal of Software Maintenance: Research and Practice 7 (1995) 317–331.

[8] J.D. Choi, J. Ferrante, Static slicing in the presence of goto statements, ACM Transactions on Programming Languages and Systems 16(4) (1994) 1097–1113.

[9] A. Cimitile, A. De Lucia, M. Munro, Qualifying reusable functions using symbolic execution, in: Proceedings of 2nd Working Conference on Reverse Engineering, Toronto, Ontario, Canada, IEEE CS Press, 1995, pp. 178–187.

[10] A. Cimitile, A. De Lucia, M. Munro, Identifying reusable functions using specification driven program slicing: a case study, in: Proceedings of International Conference on Software Maintenance, Nice, France, IEEE CS Press, 1995, pp. 124–133.

[11] A. Coen-Porisini, F. De Paoli, C. Ghezzi, D. Mandrioli, Software specialization via symbolic execution, IEEE Transactions on Software Engineering 17(9) (1991) 884–899.

[12] P.D. Coward, Symbolic execution systems—a review, Software Engineering Journal 3(6) (1988) 229–239.

[13] S. Danicic, M. Harman, Y. Sivagurunathan, A parallel algorithm for static program slicing, Information Processing Letters 56 (1995) 307–313.

[14] A. De Lucia, A.R. Fasolino, M. Munro, Understanding function behaviors through program slicing, in: Proceedings of 4th Workshop on Program Comprehension, Berlin, Germany, IEEE CS Press, 1996, pp. 9–18.

[15] J. Ferrante, K.J. Ottenstain, J. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9(3) (1987) 319–349.

[16] J. Field, G. Ramalingam, F. Tip, Parametric program slicing, in: Proceedings of 22nd ACM Symposium on Principles of Programming Languages, San Francisco, CA, ACM Press, 1995, pp. 379–392.

[17] K.B. Gallagher, J.R. Lyle, Using program slicing in software maintenance, IEEE Transactions on Software Engineering 17(8) (1991) 751–761.

[18] R. Gupta, M.J. Harrold, M.L. Soffa, An approach to regression testing using slicing, in: Proceedings of Conference on Software Maintenance, Orlando, FL, IEEE CS Press, 1992, pp. 299–308.

[19] R.J. Hall, Automatic extraction of executable program subsets by simultaneous program slicing, Journal of Automated Software Engineering 2(1) (1995) 33–53.

[20] M. Harman, S. Danicic, Y. Sivagurunathan, Program comprehension assisted by slicing and transformation, in: Proceedings of the 1st U.K. Workshop on Program Comprehension, Durham, U.K., 1995.

[21] M. Harman, S. Danicic, Amorphous program slicing, in: Proceedings of 5th International Workshop on Program Comprehension, Dearborn, MI, IEEE CS Press, 1997, pp. 70–79.

[22] M.J. Harrold, M.L. Soffa, Selecting data-flow integration testing, IEEE Software 8(2) (1991) 58–65.

[23] S. Horwitz, J. Prins, T. Reps, Integrating non-interfering versions of programs, ACM Transactions on Programming Languages and Systems 11(3) (1989) 345–387.

[24] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems 12(1) (1990) 26–60.
[25] M. Kamkar, P. Fritzson, N. Shahmerhi, Three approaches to interprocedural dynamic slicing, EUROMICRO Journal of Microprocessing and Microprogramming 38 (1993) 625–636.
[26] J.C. King, Symbolic execution and program testing, Communications of the ACM 19(7) (1976) 385–394.
[27] D.A. Kinloch, M. Munro, Understanding C programs using the combined C graph representation, in: Proceedings of International Conference on Software Maintenance, Victoria, British Columbia, Canada, IEEE CS Press, 1994, pp. 172–180.
[28] H.S. Kim, Y.R. Kwon, Restructuring programs through program slicing, International Journal on Software Engineering and Knowledge Engineering 4(3) (1994) 349–368.
[29] B. Korel, J. Laski, Dynamic slicing of computer programs, The Journal of Systems and Software 13(3) (1990) 187–195.
[30] F. Lanubile, G. Visaggio, Extracting reusable functions by flow graph-based program slicing, IEEE Transactions on Software Engineering 23(4) (1997) 246–259.
[31] J.Q. Ning, A. Engberts, W. Kozaczynski, Recovering reusable components from legacy systems by program segmentation, in: Proceedings of 1st Working Conference on Reverse Engineering, Baltimore, MD, IEEE CS Press, 1993, pp. 64–72.
[32] L. Ott, J. Thuss, The relationship between slices and module cohesion, in: Proceedings of 11th International Conference on Software Engineering, IEEE CS, 1989, pp. 198–204.
[33] K.J. Ottenstain, L.M. Ottenstain, The program dependence graph in a software development environment, ACM SIGPLAN Notices 19(5) (1984) 177–184.
[34] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (1995) 121–189.
[35] G.A. Venkatesh, The semantic approach to program slicing, ACM SIGPLAN Notices 26(6) (1991) 107–119.
[36] M. Weiser, Programmers use slices when debugging, Communications of the ACM 25 (1982) 446–452.
[37] M. Weiser, Program slicing, IEEE Transactions on Software Engineering SE-10(4) (1984) 352–357.