Understanding Function Behaviors through Program Slicing

Andrea De Lucia and Anna Rita Fasolino

Dep. of "Informatica e Sistemistica", University of Naples "Federico II", Via Claudio 21, 80125 Naples, Italy

(delucia/fasolino)@nadis.dis.unina.it

Malcolm Munro

Centre for Software Maintenance, University of Durham, South Road, DH1 3LE Durham, UK

Malcolm.Munro@durham.ac.uk

Abstract

We present conditioned slicing as a general slicing framework for program comprehension. A conditioned slice consists of a subset of program statements which preserves the behavior of the original program with respect to a set of program executions. The set of initial states of the program that characterize these executions is specified in terms of a first order logic formula on the input variables of the program. Conditioned slicing allows a better decomposition of the program giving the maintainer the possibility to analyze code fragments with respect to different perspectives. We also show how slices produced with traditional slicing methods can be reduced to conditioned slices. Conditioned slices can be identified by using symbolic execution techniques and dependence graphs.

1 Introduction

The comprehension of an existing software system consumes from 50% up to 90% of its maintenance time. Comprehending a software system can be defined as the process of abstracting higher level descriptions of the system - which employ typical application domain concepts and terms - from lower level descriptions, like control-flow/data-flow oriented documents. The goal of the abstraction process is therefore the production of a software model that includes objects and inter-relations from the real world domain, while omitting less significant details of the programming domain. For years researchers have devoted their efforts trying to understand how programmers comprehend code and several program understanding models have been proposed. von Mayrhauser and Vans in [23] provide a useful survey about six cognition models, compare them, and identify the their key features.

The common feature of all cognition models is that they employ existing knowledge to produce new knowledge about the mental model of the software under consideration. Both a 'technical' knowledge (knowledge of programming language, environment, techniques, models) and a 'semantic' knowledge (application domain expertise) are used during the process. This knowledge is exploited during the comprehension process for reconstructing the mapping between software descriptions at different abstraction levels. All models agree that comprehension proceeds either top-down, or bottom-up, or a combination of these two. The model developed by von Mayrhauser and Vans integrates the former models as components.

The *top-down model* of program comprehension is typically invoked when code under consideration is familiar. In this case a domain knowledge is available, therefore a description of the conceptual components of the application domain and of the way they interact is provided. The programmer understands code by exploiting this knowledge to formulate hypotheses about the meaning of the program segments being analyzed. Each hypothesis must be confirmed by scanning code for *beacons*. Beacons consist of pieces of code implementing typical data structures and algorithms that the programmer recognizes and correctly associates to its current hypotheses. Hypotheses are iteratively refined, producing new sub-goals to be verified by scanning code again. The process halts whenever each component of the application domain has been identified in code.

The *bottom-up model* of program comprehension is vice-versa invoked when the code under consideration is completely new to the programmer. The first mental representation of the program she builds is a control-flow abstraction called the program model. The program model is created via the chunking of microstructures into macrostructures and via cross-referencing. Starting from the program model, a further model can be abstracted which maps the control-flow knowledge about code to the real world domain knowledge. The generation of this model proceeds by associating single program objects of the program model (like statements, data, blocks of statements, subroutines, and so on) with actions and entities of the real world. The process continues by formulating new hypotheses to aggregate these plans into higher order plans.

The *integrated model* by von Mayrhauser and Vans [23] is based on the idea that code comprehension involves both top-down and bottom-up activities. The process does not proceed either in the top-down direction, or in the bottom-up, but rather continuously switches between these two approaches. The integrated model includes four main components, that are the top-down model, the situation model, the pro-

gram model, and the knowledge base. The first three constitute comprehension processes while the fourth is needed to reconstruct the first three. The knowledge base stores any new and inferred knowledge, that is used to produce the other models.

The common feature of all these models consists of the iterative mechanism of formulating hypotheses and validating (or refusing) them. While formulating hypotheses always requires domain knowledge and expertise, validating them essentially means scanning the code looking for significant beacons. This can be an expensive task : software is a complex artifact, often composed of different parts interconnected and interacting in complex ways. Furthermore, such interactions are sometimes delocalized and, as Letovsky and Soloway [18] have established, programmers have difficulty in understanding code with non local interactions. When they scan code, programmers implement several tasks which span from *tracing* to *chunking*, from *slicing* and *data-flow analysis* to *functional and calling dependencies analysis*. All these tasks are needed in order to dominate the complexity of software artifacts. Chunking, for instance, is an abstraction mechanism used in bottom-up approaches which allows code chunks to be associated with more abstract descriptions. Code chunks are grouped together to form larger chunks, until the entire program is understood. In this way a hierarchical internal semantic representation of the program is built from the bottom-up.

A technique that programmers may use when scanning code for beacons is program slicing. In the original Weiser's definition, program slicing consists of finding all statements in a program that directly or indirectly affect the value of a variable occurrence. This leads to a subset of program statements - the slice - that captures some subset of the program behavior. The slice isolated is easier to be analyzed than the original program as it represents a sub-component of the whole program. Two main slicing definitions have been introduced in literature, static slicing [25] and dynamic slicing [16]. These techniques have been successfully employed for program comprehension during different maintenance tasks, like program analysis, testing, debugging. While static slicing is useful for isolating and supporting the comprehension of code implementing a functionality, dynamic slicing has been used in debugging for identifying the statements affecting the value of a variable on a program execution that reveals an incorrect behavior. However, for code implementing a complex functionality which behaves differently depending on the input to the program, static slicing could produce slices that are too large and difficult to understand, while dynamic slicing usually produce slices that can result too simple and not significant for the comprehension process.

Different definitions of slicing have been proposed in the literature for specifying program slices that are correct with respect to a set of input to the program. For example, *quasi-static slicing* [22] assigns a fixed value to a subset of the input variables and analyzes the program behavior while the other input variables vary. *Simultaneous dynamic slicing* [11] combines the use of a set of test-cases with program slicing: it extends and simultaneously applies to a set of test-cases the dynamic slicing technique, thus selecting program statements corresponding to a particular program behavior observed from specific test-cases. Quasi static slicing and simultaneous dynamic slicing use two different approaches to specify a set of initial states of the program with respect to which the behavior of the function can be observed. However, some function behavior could be characterized by relations between input values that cannot be expressed by a prefix of the input or by a set of test cases. In order to identify program slicing corresponding to any function behavior, a more general model which allow the specification of any initial state of the program is required. This can be done using a first order logic formula which maps a subset of the input program variables onto a set of initial states to the program. We call *conditioned slice* the slice obtained by adding such a condition on the input variables to the slicing criterion [5].

In this paper the role of conditioned slicing as a general program comprehension framework that includes all slicing paradigms is described. In section 2 static and dynamic slicing are recalled by describing their use in program comprehension. Section 3 outlines the need in program comprehension for identifying function behaviors with respect to a set of input to the program. In section 4 a formal definition of conditioned slicing is presented and its use as general slicing framework is outlined. Techniques for finding conditioned slices are also introduced. Concluding remarks are discussed in section 5.

2 Program Slicing

Program slicing has been introduced by Weiser [25] as a program decomposition technique based on the analysis of the control and data flow. Experimental studies show that most programmers try to identify program bugs by using slices of the program composed of statements which affect the computation of interest [24]. A survey about program slicing techniques and their applications can be found in [21].

In this section we describe two basic approaches to program slicing, called static slicing [25] and dynamic slicing [16]. The difference between them is that a static slice is defined with respect to all the execution paths of the program (both feasible and infeasible), while a dynamic slice only takes into account a particular execution path obtained from one input to the program.

2.1 Background

A one-entry/one-exit program can be modeled as a graph, whose nodes represent program statements and whose edges represent transfer of the control. In this section some basic definitions about flowgraph analysis are recalled.

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* from node n to node m of length k is a list of nodes $\langle p_1, p_2, \ldots, p_k \rangle$ such that $p_1 = n$, $p_k = m$, and $\forall i, 1 \leq i \leq k-1, (p_i, p_{i+1}) \in E$.

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 $E^{-1} = \{(m, n) \mid (n, m) \in E\}$.

In the following we will associate any one-entry/one-exit program P with its set of variable V and a hammock graph $HG = (N, E, n_0, n_e)$.

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 variables in V assume before the execution of each statement. Such a mapping will be referred to as *state trajectory* [25]. 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 list of ordered pairs $T = \langle (p_1, \sigma_1), (p_2, \sigma_2), \ldots, (p_k, \sigma_k) \rangle$, where $p_i \in N$, $1 \leq i \leq k$, $\langle p_1, p_2, \ldots, p_k \rangle$ is a path from n_0 to n_e, and σ_i, $1 \leq i \leq k$, is a function mapping the variables in V to the values they assume immediately before the execution of p_i.

2.2 Static Slicing

Weiser [25] defines a static program slice as any executable subset of program statements which preserves the behavior 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 $T = \langle (p_1, \sigma_1), (p_2, \sigma_2), \ldots, (p_k, \sigma_k) \rangle$ a state trajectory of P on input I. $\forall i, 1 \leq i \leq k$:

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

where $\sigma_i \mid V$ is σ_i restricted to the domain V, and λ is the empty string. The extension of Proj′ 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:

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

[1] We assume program termination.

A program slice is therefore defined behaviorally as any subset of a program which preserves a specified projection of its behavior.

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' that is obtained from P by deleting zero or more statements, and 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 [25], because it requires that the instruction p always appears in the static slice. This is not a limitation, in particular if program slicing is used for program comprehension. Indeed, programmers can be easily confused if the instruction p of the slicing criterion is not included in the slice, particularly if p is in a loop [16].

As an example of static slice, let us consider the program in Figure 1. The static slice on the slicing criterion $C = (32, \{\texttt{sum}\})$[2] is shown in Figure 2.

Although the problem of finding minimal static slices is *undecidable*, Weiser proposes an iterative algorithm [25] based on data flow and on the *influence* of predicates on statement execution, which compute conservative slices, guaranteed to have the properties of the definition above. The slice is computed as the set of all statements of the program that might affect directly or indirectly the value of the variable in V just before the execution of p. Program slices can also be computed using the program dependence graph [10] both at intraprocedural [20] and interprocedural level [12]. An enhanced slicing algorithm based on dependence graphs [6] also allows the computation of correct slices in the presence of goto statements.

Static slicing can be used in program comprehension to identify the subset of a program corresponding to a functionality [8, 17, 19]. In this case, the set of variables V in the slicing criterion corresponds to the set of output variables of the function, while the statement p corresponds to the last statement of the function. The process of identifying a slicing criterion requires the knowledge of the data model and how it has been traced onto the program variables. Whenever this is not available, human knowledge and expertise is required to abstract it from code. Also, the identification of the statement of the slicing criterion is based on code analysis. Some authors proposed different definitions of slice that include in the slicing criterion the set of input variables of the function [17] or an initial statement [8], in order to stop the computation of the slice whenever the code implementing the expected function has been identified.

2.3 Dynamic Slicing

Program slicing has been first proposed as a tool for decomposing programs during debugging, in order to allow a better understanding of the portion of

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

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

Figure 1: Example program

code which revealed an error [24, 25]. In this case the slicing criterion contains the variables which produced an unexpected result on some input to the program. However, a static slice very often contains statements which have no influence on the values of the variables of interest for the particular execution in which the anomalous behavior of the program was discovered.

Korel and Lasky [16] 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, allowing a better understanding of the code and easier localization of the bugs. Another advantage of dynamic slicing with respect to the static approach is the run-time handling of arrays and pointer variables. While in the static slicing each definition or use of any array element is treated as a definition or use of the entire array (because of the difficulty of determining the values of array subscripts),

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

Figure 2: Example static slice

in dynamic slicing any array element can be individually treated, so allowing to further reduce the size of the slice. Moreover, it is possible to determine which objects are pointed to by pointer variables during program execution.

From a formal point of view, a dynamic slice is defined with respect to a particular trajectory [16]. In this case, the slicing criterion refers to a statement in a particular position in the state trajectory. However, we will always refer to the last occurrence of a statement in a trajectory. In this way the only difference between static and dynamic slicing is that a dynamic slice is required to preserve the behavior of the original program on only one input, where the static slice must be correct on any input. In another work [11] this is implicitly assumed, while Agrawal and Horgan [1] compute dynamic slices with respect to the last statement of the program. A *backward* slice (both static and dynamic) is also considered with respect to the last statement in the semantic approach to program slicing [22].

Definition 2.8 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 2.9 A *dynamic slice* of a program P on a dynamic slicing criterion $C = (I, p, V)$ is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and whenever P halts on input I with state trajectory T, then P' also halts on input I with state trajectory

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

Figure 3: Example dynamic slice

T', and $\text{Proj}_{(p,V)}(T) = \text{Proj}_{(p,V)}(T')$.

For example, let us consider the program in Figure 1. Figure 3 shows the dynamic slice on the slicing criterion $C = (I, 32, \{\texttt{sum}\})$, where $I = \langle \texttt{test0} = 0, \texttt{n} = 2, \texttt{a}_1 = 0, \texttt{a}_2 = 2 \rangle$[3].

Korel and Lasky [16] propose an iterative algorithm based on dynamic data flow and control influence to compute a subtrajectory T' of T that meets the definition, and from which the slice can be reconstructed. The algorithm also requires that if any occurrence of a statement (within a loop) in the trajectory is included in the slice, then all other occurrences of that statement be automatically included in the slice. This requirement ensures that the slice extracted is executable. Other approaches based on dynamic dependence graphs [1, 13] produce more refined slices, considering only the occurrences of statements in the trajectory that affect the computation of the variables in the slicing criterion. The resulting slice is not necessarily an executable subset of the original program, but it is still useful for the purposes of program understanding during debugging and data-flow testing.

3 Understanding Function Behaviors

The traditional static and dynamic slicing models consider subsets of the program with respect to either all possible executions or just one execution, respectively. However, a program function can behave differently depending on particular inputs to the program. For example, the price and the insurance costs for renting a car might depend on the type of the car, on the age of the client and on the duration of the renting period. Whenever the knowledge about the application domain is available, the maintainer can attempt to understand these different function behaviors, by isolating the portion of code corresponding to each of such behaviors. Hypotheses about the conditions that characterize these behaviors can be formulated and expressed in terms of particular inputs to the program.

Traditional static slicing techniques might isolate too large code components which include all the possible behaviors of the function. On the other hand, dynamic slicing is only suitable to identify a code fragment with respect to one input: the produced slice could be too small to be generalized and considered as the code fragment implementing the desired behavior. In order to capture a set of program executions corresponding to a particular behavior of a program, two variants of slicing have been proposed in the literature. The first method, called *quasi static slicing* [22], specifies the set of initial states of the program by assigning an initial value to a subset of the input variables. The second method, called *simultaneous dynamic program slicing* [11], explicitly uses a set of test cases, each of which corresponds to an initial state.

3.1 Quasi Static Slicing

Venkatesh [22] proposes a notion of slice that falls between static and dynamic slices. The motivation of such a slice, called *quasi-static* slice arises from applications in which values of some inputs are fixed while the behavior of the program must be analyzed and understood when other input values vary.

Definition 3.1 Let V_{in} be the set of input variables of a program P and $V'_{in} \subseteq V_{in}$. Let I' be an input for the variables in V'_{in}. A *quasi static slicing criterion* of a program P is a quadruple $C = (V'_{in}, I', p, V)$, where p is a statement in P and V is a subset of the variables in P.

Definition 3.2 Let V_{in} be the set of input variables of a program P and $V'_{in} \subseteq V_{in}$. Let I' be an input for the variables in V'_{in}. A *completion* I of I' is any input for the variables in V_{in}, such that $I' \subseteq I$.

Each completion I of I' identifies a trajectory T. We can associate I' with the set of trajectories that are produced by its completions. A quasi static slice is any subset of the program which reproduces the original behavior on each of these trajectories.

Definition 3.3 A *quasi static slice* of a program P on a quasi static slicing criterion $C = (V'_{in}, I', p, V)$ is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and whenever P halts on input I, such that $I' \subseteq I$, with state trajectory T, then P' also halts on input I with state trajectory T', and $\text{Proj}_{(p,V)}(T) = \text{Proj}_{(p,V)}(T')$.

It is straightforward to see that quasi static slicing includes both the static and dynamic paradigms. Indeed, when the set of variables V'_{in} is empty, quasi static slicing reduces to static slicing, while for $V'_{in} = V_{in}$ a quasi static slice coincides with a dynamic slice.

As an example, let us consider the program in Figure 1. The quasi static slice on the slicing criterion $C = (\{\texttt{test0}\}, 0, 32, \{\texttt{sum}\})$ is shown in Figure 4.

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

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

Figure 4: Example quasi static slice

```
1  main() {
2    int a, test0, n, i, possum, negsum, sum;
3    scanf("%d", &test0); scanf("%d", &n);
     scanf("%d", &a);
4    i = 1;
5    possum = negsum = 0;
6    while (i <= n && a <= n) {
7        if  (a > 0)
8             possum += a;
10       else if (a < 0) {}
13            else if (test0)
14                      if (possum >= negsum)
15                           possum = 0;
20      i++;
21      scanf("%d", &a); }
22   if (i <= n)  { }
26   else if (possum >= negsum)
27              sum = possum;
32   printf("%d \n", sum);
```

Figure 5: Example simultaneous dynamic 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 [2], a technique to specialize programs with respect to partial input. Partial evaluation has been used for the comprehension of Fortran programs [3]. By specifying the values of some of the input variables, techniques such as constant propagation and simplification can be used to attempt to reduce expressions to constants. In this way, the values of some program predicates can be evaluated, allowing not executed branches (for the particular partial input) to be discarded. The reduced program is easier to understand than the original one, because details which are not interesting for the particular computations under consideration are eliminated.

Partial evaluation can be considered a complementary technique to program slicing, aiming to understand old programs which have become very complex due to extensive maintenance [3]. Quasi static slicing has the characteristic to combine together program slicing and partial evaluation, so allowing a better reduction of the program with respect to the slicing criterion.

3.2 Simultaneous Dynamic Slicing

A different approach to the definition of a slice with respect to a set of executions of the program has been proposed by Hall [11]. This new slicing technique combines the use of a set of test cases with program slicing. The method is called *simultaneous dynamic program slicing* because it extends and simultaneously applies to a set of test cases the *dynamic slicing* technique [16] which produces executable slices that are correct on only one input.

Definition 3.4 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, $1 \leq j \leq m$, 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 3.5 A *simultaneous dynamic slice* of a program P on simultaneous dynamic slicing criterion $C = (\{I_1, I_2, \ldots, I_m\}, p, V)$ is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and whenever P halts on input I_j, $1 \leq j \leq m$, 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, let us consider the program in Figure 1. Figure 5 shows the simultaneous dynamic slice on the slicing criterion $C = (I_1, I_2, 32, \{\texttt{sum}\})$, where $I_1 = \langle \texttt{test0} = 0, \texttt{n} = 2, \texttt{a}_1 = 0, \texttt{a}_2 = 2 \rangle$ and $I_2 = \langle \texttt{test0} = 1, \texttt{n} = 2, \texttt{a}_1 = 0, \texttt{a}_2 = 2 \rangle$.

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 [11]. An iterative algorithm is presented [11] that, starting from an initial set of statements, incrementally constructs the simultaneous dynamic slice, by computing at each iteration a larger dynamic slice.

This approach can be used in program comprehension for the isolation of the subset of the statements corresponding to a particular program behavior. It can be considered a refinemenrt of the method proposed by Wilde *et al.* [26] that consider the problem of locating functionalities in code as the identification of the relation existing between the ways the user and the programmer see the program. From the user point of view, a pro-

gram consists of a collection of, possibly overlapping, functionalities $FUNCS = \{f_1, f_2, ..., f_N\}$ while the programmer's view consists of a collection of program components $COMPS = \{c_1, c_2, ..., c_M\}$. The problem is the identification of the components in $COMPS$ which contribute to implement a functionality in $FUNCS$, i.e., the construction of a relation $IMPL \subseteq COMPS \times FUNCS$. The link between components and functionalities may be provided by test cases. A test case T_i exhibits a set of functionalities $F(T_i) = \{f_{i,1}, f_{i,2}, ...\}$ which can be identified by a system user. On the other hand, a test case also exercises a set of program components $C(T_i) = \{c_{i,1}, c_{1,2}, ...\}$ which can be identified by instrumenting the code and monitoring its execution.

Both *deterministic* and *probabilistic* techniques have been proposed to analyze the traces resulting from the program execution [26]. However, while these approaches are cost-effective, very practical and easy to implement and use, they are only good to find components that are unique to a particular functionality. In general, the method lacks in precision, because the software component identified could be too large and include more functionalities than the one sought. Simultaneous dynamic slicing can be considered as a refinement of methods for localizations of functions based on test cases [26], because it takes into account the data flow of the program and then allows the reduction of the set of selected statements.

4 Conditioned Slicing

Quasi static slicing and simultaneous dynamic slicing use two different approaches to specify a set of initial states of the program with respect to which observe the behavior of the function. In simultaneous dynamic slicing the set of initial states is finite, while for quasi static slicing it might be infinite. However, the quasi static slicing paradigm does not include simultaneous dynamic slicing, because it only uses a prefix of the input and does not consider a finite set of values for a variable. On the contrary, some function behavior could be characterized by relations between input values that cannot be expressed by a prefix of the input. For example, it would be desirable to observe the program whenever the value of some input variables falls in a (possibly infinite) range of suitable values. As further example, the transaction for moving money from or to an account could considerably depend on the relation between the amount to be moved and the total amount of the account. Such relations can be expressed neither as a prefix of the input nor as a (finite) set of test cases.

In order to identify program slices corresponding to any function behavior, we need a more general model which allows to specify any initial state of the program. This can be done using a first order logic formula which maps a subset of the input program variables onto a set of initial states to the program. The slice resulting by adding such a condition on the input variables to the slicing criterion will be referred to as *conditioned slice*. Conditioned slicing is able to identify slices with respect to any subset of program

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

Figure 6: Example conditioned slice

executions. In the following we will show how this model can be used as a framework to switch between different slicing paradigms.

Definition 4.1 Let V_{in} be the set of input variables of a program P, $V'_{in} \subseteq V_{in}$ and F' be a first order logic formula on the variables in V'_{in}. A *conditioned slicing criterion* of a program P is a quadruple $C = (V'_{in}, F', p, V)$, where p is a statement in P and V is a subset of the variables in P.

The formula F' identifies a set of input to the program and consequently a set of state trajectories.

Definition 4.2 Let V_{in} be the set of input variables of a program P, $V'_{in} \subseteq V_{in}$ and F a first order logic formula on the variables in V'_{in}. Let $IS'(F)$ be the set of input I' for V'_{in} that satisfies the formula F. The *input set* $IS(F)$ of P with respect to F is the set of input I to the program such that I is a completion of some $I' \in IS'(F)$.

Each $I \in IS(F)$ identifies a trajectory T. A conditioned slice is any subset of the program which reproduces the original behavior on each of these trajectories.

Definition 4.3 A *conditioned slice* of a program P on a conditioned slicing criterion $C = (V'_{in}, F, p, V)$ is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and whenever P halts on input I, where $I \in IS(F)$, 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, consider the program in Figure 1. The conditioned slice on the slicing criterion $C = (V'_{in}, F, 32, \{\mathtt{sum}\})$, with $V'_{in} = \{\mathtt{n}\} \bigcup_{1 \le i \le n} \{\mathtt{a}_i\}$ and $F = (\forall i,\ 1 \le i \le n,\ \mathtt{a}_i > 0)$, is shown in Figure 6.

Notice that any slicing criterion can be expressed as a conditioned slicing criterion. Therefore, conditioned slicing can be used as a general framework including all slicing methods. Any slice can be identified by suitably specifying a set of variables, a program point and a condition on the program input corresponding to a set of initial states. For example, the static slicing criterion $(32, \{\texttt{sum}\})$ can be expressed as $(\emptyset, \mathit{true}, 32, \{\texttt{sum}\})$ and the dynamic slicing criterion $(I, 32, \{\texttt{sum}\})$, where $I = \langle \texttt{test0} = 0, \texttt{n} = 2, \texttt{a}_1 = 0, \texttt{a}_2 = 2 \rangle$, is $(V'_{in}, F, 32, \{\texttt{sum}\})$, where $V'_{in} = \{\texttt{test0}, \texttt{n}, \texttt{a}_1, \texttt{a}_2\}$ and[4] $F = (\texttt{test0} = 0 \wedge \texttt{n} = 2 \wedge \texttt{a}_1 = 0 \wedge \texttt{a}_2 = 2)$. The quasi static criterion $(\{\texttt{test0}\}, 0, 32, \{\texttt{sum}\})$ can be expressed as $(\{\texttt{test0}\}, \texttt{test0} = 0, 32, \{\texttt{sum}\})$; the simultaneous slicing criterion $(\{I_1, I_2\}, 32, \{\texttt{sum}\})$, where $I_1 = \langle \texttt{test0} = 0, \texttt{n} = 2, \texttt{a}_1 = 0, \texttt{a}_2 = 2 \rangle$ and $I_2 = \langle \texttt{test0} = 1, \texttt{n} = 2, \texttt{a}_1 = 0, \texttt{a}_2 = 2 \rangle$, is $(V'_{in}, F, 32, \{\texttt{sum}\})$, where $V'_{in} = \{\texttt{test0}, \texttt{n}, \texttt{a}_1, \texttt{a}_2\}$ and $F = ((\texttt{test0} = 0 \wedge \texttt{n} = 2 \wedge \texttt{a}_1 = 0 \wedge \texttt{a}_2 = 2) \vee (\texttt{test0} = 1 \wedge \texttt{n} = 2 \wedge \texttt{a}_1 = 0 \wedge \texttt{a}_2 = 2))$

Conditioned slicing was first introduced by Canfora *et al.* [5] to isolate function behaviors in code for software reuse. Algorithms based on the program dependence graphs heve also been provided for extracting a conditioned slice. However, human interaction is intensively required to trace the condition of the slicing criterion into the program variables and predicates. The method can be improved by using formal method tools, such as symbolic executors [14, 9] and theorem provers, e.g. [4].

4.1 Finding Conditioned Slices

Intuitively, a conditioned slice can be identified 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 can be used to compute the reduced program, also called *conditioned program* in [5].

While in traditional execution the values of program's 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 state in traditional execution, a *symbolic state* is a pair $\langle \mathit{State}, PC \rangle$, where *State* is a set of pairs of the form $\langle M, \alpha \rangle$, M and α being a memory location and its symbolic value respectively, and PC is a first order logic formula called *path-condition*.

The path-condition represents the condition which must be satisfied in order for an execution to follow the particular associated path on the control flow. Indeed, while the evaluation of a predicate in traditional execution unequivocally allows the selection of the branch to follow, the symbolic evaluation of a predicate might generate two possible executions. For example, the execution of the predicate in line 7 of the program in Figure 1, in the symbolic state:

$$\begin{aligned}\langle S_1, P_1 \rangle \;=\; & \langle \{\langle \texttt{a}, \alpha_1 \rangle, \langle \texttt{test0}, \beta \rangle, \langle \texttt{n}, \gamma \rangle, \langle \texttt{i}, 1 \rangle, \\ & \langle \texttt{posprod}, 1 \rangle, \langle \texttt{negprod}, 1 \rangle, \\ & \langle \texttt{possum}, 0 \rangle, \langle \texttt{negsum}, 0 \rangle, \\ & \langle \texttt{prod}, \mathit{undef} \rangle, \langle \texttt{sum}, \mathit{undef} \rangle, \}, \\ & 1 \leq \gamma \wedge \alpha_1 \leq \gamma \rangle\end{aligned}$$

produces the two symbolic states:

$$\begin{aligned}\langle S_2, P_2 \rangle \;=\; & \langle \{\langle \texttt{a}, \alpha_1 \rangle, \langle \texttt{test0}, \beta \rangle, \langle \texttt{n}, \gamma \rangle, \langle \texttt{i}, 1 \rangle, \\ & \langle \texttt{posprod}, 1 \rangle, \langle \texttt{negprod}, 1 \rangle, \\ & \langle \texttt{possum}, 0 \rangle, \langle \texttt{negsum}, 0 \rangle, \\ & \langle \texttt{prod}, \mathit{undef} \rangle, \langle \texttt{sum}, \mathit{undef} \rangle, \}, \\ & 1 \leq \gamma \wedge \alpha_1 \leq \gamma \wedge \alpha_1 > 0 \rangle\end{aligned}$$

$$\begin{aligned}\langle S_3, P_3 \rangle \;=\; & \langle \{\langle \texttt{a}, \alpha_1 \rangle, \langle \texttt{test0}, \beta \rangle, \langle \texttt{n}, \gamma \rangle, \langle \texttt{i}, 1 \rangle, \\ & \langle \texttt{posprod}, 1 \rangle, \langle \texttt{negprod}, 1 \rangle, \\ & \langle \texttt{possum}, 0 \rangle, \langle \texttt{negsum}, 0 \rangle, \\ & \langle \texttt{prod}\ \mathit{undef} \rangle, \langle \texttt{sum}, \mathit{undef} \rangle, \}, \\ & 1 \leq \gamma \wedge \alpha_1 \leq \gamma \wedge \alpha_1 \leq 0 \rangle\end{aligned}$$

Notice that the two path-conditions carry the conditions that must be satisfied in order to follow the corresponding branches, respectively. The path-condition can be used to discard infeasible paths. This is in particular true whenever the symbolic execution of a program is made with an initial non trivial path-condition. Indeed, let us suppose to symbolically execute the program in Figure 1 with the initial path-condition[5] $\forall i, 1 \leq i \leq \gamma, \alpha_i > 0$, where α_i is the input symbolic value for $\texttt{a}_i$, $1 \leq i \leq \gamma$, and γ is the input symbolic value for $\texttt{n}$. The symbolic state before the execution of the predicate in line 7 will be:

$$\begin{aligned}\langle S'_1, P'_1 \rangle \;=\; & \langle \{\langle \texttt{a}, \alpha_1 \rangle, \langle \texttt{test0}, \beta \rangle, \langle \texttt{n}, \gamma \rangle, \langle \texttt{i}, 1 \rangle, \\ & \langle \texttt{posprod}, 1 \rangle, \langle \texttt{negprod}, 1 \rangle, \\ & \langle \texttt{possum}, 0 \rangle, \langle \texttt{negsum}, 0 \rangle, \\ & \langle \texttt{prod}\ \mathit{undef} \rangle, \langle \texttt{sum}, \mathit{undef} \rangle, \}, \\ & (\forall i, 1 \leq i \leq \gamma, \alpha_i > 0) \wedge 1 \leq \gamma \wedge \alpha_1 \leq \gamma \rangle\end{aligned}$$

Due to the initial condition, the path-condition P'_1 implies the condition $\alpha_1 > 0$ obtained from the evaluation of the predicate in line 7 in the state S'_1. Therefore, the *false* branch can be discarded and the execution will continue on the *true* branch with symbolic state unchanged.

The evaluation of such implications is in general an *undecidable* problem and requires human interaction. However, in most cases the symbolic expressions can be simplified and these implications can be automatically evaluated by a theorem prover. Figure 7 shows the conditioned program resulting from the symbolic

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

[5] The initial path-condition corresponds to the condition $F = (\forall i, 1 \leq i \leq n, \texttt{a}_i > 0)$ of the example slicing criterion, from which the conditioned slice in Figure 6 is obtained.

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

Figure 7: Example conditioned program

execution of the program in Figure 1, with the initial path-condition. Notice that whenever the condition $F = (\forall i,\ 1 \leq i \leq n,\ \mathtt{a}_i > 0)$ 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 neither the loop condition nor its negation. While in general a solution for this problem requires the determination of a suitable *loop invariant* which allows to continue the symbolic execution at the end of the loop (see [7] for a survey of such techniques), 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 the user that can decide the number of time the loop must be executed. This interaction also allows a better comprehension of the program. Indeed, whenever the user chooses the path to follow, the path-condition is modified accordingly. For example, for the *while* loop of the program in Figure 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 generalize the sample executions and recover a loop invariant for the sake of precision.

Dependence graphs have been used to construct both static [20, 12] and dynamic [1] slices. On the same line, dependence graphs can be used to compute conditioned slices. A first simple solution consists of marking all the statements of a dependence graph that are symbolically executed with the initial path-condition. Then a conditioned slice can be computed by backward traversing control and data dependence edges between marked edges and including in the slice all the statements and predicates reached by this transitive closure [5].

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

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

From a static point of view, the def-use dependencies (1, 4, `x`) and (3, 4, `x`) are represented on a program dependence graph. Therefore, a static slice on the slicing criterion (4, {`z`}) will include the whole code fragment. On the contrary, a conditioned slice on the slicing criterion (V_{in}, 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, the technique described above will consider all the static dependencies between marked nodes of the dependence graph and therefore will include the statement in line 1 in the conditioned slice.

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. As an example, the symbolic execution of the code fragment above, will mark all the statements and the data dependency (3, 4, `x`), but not the data dependency (1, 4, `x`). This allows to discard the statement in line 1 that cannot be reached on marked edges.

5 Conclusion

In this paper the role of conditioned slicing as a general framework for program comprehension that includes all slicing paradigms has been addressed. The main slicing techniques introduced in the literature, static slicing [25], dynamic slicing [16], quasi-static slicing [22], and simultameous dynamic slicing [11] have been considered and their limitations in supporting the comprehension of code implementing complex functionalities have been outlined. Because such functionalities behave differently depending on the input to the program, a more general slicing model is required that allows to identify program slices corresponding to any function behavior. Conditioned slicing allows to specify any set of initial states of the program by using a first order logic formula.

Program slices can then be identified with respect to any subset of program executions. A conditioned slice can be extracted by first simplifying the program

with respect to the condition on the input, and then computing the slice on the reduced program. A symbolic executor [9, 14] can be used to compute the reduced program, while program dependencies are exploited for isolating the program slice. We are currently implementing conditioned slicing for C programs in Prolog environment. A fine-grained representation for C programs, called the Combined C Graph (CCG) [15] (consisting of an abstract syntax tree combined with a control flow graph and program dependencies) is used to perform both symbolic execution and program slicing [7, 15].

References

[1] H. Agrawal and J.R. Horgan, "Dynamic program slicing", in *Procs of ACM SIGPLAN Conf. on Progr. Lang. Design and Implem.*, White Plains, New York, U.S.A., 1990, pp. 246-256.

[2] D. Bjørner, A.P. Ershov, and N.D. Jones, *Partial evaluation and mixed computation*, North-Holland, 1987.

[3] S. Blazy and P. Facon, "Partial evaluation as an Aid to the Comprehension of Fortran Programs", in *Procs of 2nd Work. on Progr. Compr.*, Capri, Italy, IEEE CS Press, 1993, pp. 46-54.

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

[5] G. Canfora, A. Cimitile, A. De Lucia, and G.A. Di Lucca, "Software salvaging based on conditions", in *Procs of Intern. Conf. on Soft. Maint.*, Victoria, British Columbia, Canada, IEEE CS Press, 1994, pp. 424-433.

[6] J.D. Choi and J. Ferrante, "Static slicing in the presence of goto statements", *ACM Trans. on Progr. Lang. and Syst.*, vol. 16, no. 4, July 1994, pp. 1097-1113.

[7] A. Cimitile, A. De Lucia, and M. Munro, "Qualifying reusable functions using symbolic execution", in *Procs of 2nd Work. Conf. on Rev. Eng.*, Toronto, Ontario, Canada, IEEE CS Press, 1995, pp. 178-187.

[8] A. Cimitile, A. De Lucia, and M. Munro, "Identifying reusable functions using specification driven program slicing: a case study", in *Procs of Intern. Conf. on Soft. Maint.*, Nice, France, IEEE CS Press, 1995, pp. 124-133.

[9] P.D. Coward, "Symbolic execution systems – a review", *Soft. Eng. Jour.*, vol. 3, no. 6, Nov. 1988, pp. 229-239.

[10] J. Ferrante, K.J. Ottenstain, and J. Warren, "The program dependence graph and its use in optimization", *ACM Trans. on Progr. Lang. and Syst.*, vol. 9, no. 3, July 1987, pp. 319-349.

[11] R.J. Hall, "Automatic extraction of executable program subsets by simultaneous program slicing", *Jour. of Autom. Soft. Eng.*, vol. 2, no. 1, Mar. 1995, pp. 33-53.

[12] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs", *ACM Trans. on Progr. Lang. and Syst.*, vol. 12, no. 1, Jan. 1990, pp. 26-60.

[13] M. Kamkar, P. Fritzson, and N. Shahmerhi, "Three approaches to interprocedural dynamic slicing", *EUROMICRO Jour.*, vol. 38, 1993, pp. 625-636.

[14] J.C. King, "Symbolic execution and program testing", *Comm. of the ACM*, vol. 19, no. 7, July 1976, pp. 385-394.

[15] D.A. Kinloch and M. Munro, "Understanding C programs using the combined C graph representation", in *Procs of Intern. Conf. on Soft. Maint.*, Victoria, British Columbia, Canada, IEEE CS Press, 1994, pp. 172-180.

[16] B. Korel and J. Laski, "Dynamic slicing of computer programs", *The Jour. of Syst. and Soft.*, vol. 13, no. 3, Nov. 1990, pp. 187-195.

[17] F. Lanubile and G. Visaggio, "Function recovery based on program slicing", in *Proceedings of Intern. Conf. on Soft. Maint.*, Montreal, Quebec, Canada, IEEE CS Press, 1993, pp. 396-404.

[18] S. Letovsky and E. Soloway, "Delocalized plans and program comprehension", *IEEE Software*, vol. 3, no. 3, 1986, pp. 41-49.

[19] J.Q. Ning, A. Engberts, and W. Kozaczynski, "Recovering reusable components from legacy systems by program segmentation", in *Proc.s of 1st Work. Conf. on Rev. Eng.*, Baltimore, Maryland, U.S.A., IEEE CS Press, 1993, pp. 64-72.

[20] K.J. Ottenstain and L.M. Ottenstain, "The program dependence graph in a software development environment", *ACM SIGPLAN Notices*, vol. 19, no. 5, May 1984, pp. 177-184.

[21] F. Tip, "A survey of program slicing techniques", *Jour. of Progr. Lang.*, vol. 3, 1995, pp. 121-189.

[22] G.A. Venkatesh, "The semantic approach to program slicing", *ACM SIGPLAN Notices*, vol. 26, no. 6, 1991, pp. 107-119.

[23] A. von Mayrhauser, and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution", *IEEE Computer*, vol. 28, no. 8, Aug. 1995, pp. 44-55.

[24] M. Weiser, "Programmers use slices when debugging", *Comm. of the ACM*, vol. 25, July 1982, pp. 446-452.

[25] M. Weiser, "Program slicing", *IEEE Trans. on Soft. Eng.*, vol. SE-10, no. 4, July 1984, pp. 352-357.

[26] N. Wilde, J.A. Gomez, T. Gust, and D. Strasburg, "Locating user functionality in old code", in *Procs of Intern. Conf. on Soft. Maint.*, Orlando, Florida, U.S.A., IEEE CS Press, 1992, pp. 200-205.