# Building Executable Union Slices using Conditioned Slicing

**Sebastian Danicic**
Department of Computing
Goldsmiths College,
University of London,
New Cross,
London SE14 6NW, UK
s.danicic@gold.ac.uk

**Andrea De Lucia**
Dipartimento di Matematica
e Informatica
University of Salerno
Via ponte don Melillo
84084, Fisciano (SA), Italy
adelucia@unisa.it

**Mark Harman**
Department of Information
Systems and Computing,
Brunel University,
Uxbridge,
Middlesex, UB8 3PH, UK
mark.harman@brunel.ac.uk

## Abstract

*Program slicing can be used as a support for program comprehension, because it allows a large program to be divided up into smaller slices, each of which can be understood in isolation from the rest. As such, slicing facilitates the familiar approach of 'divide and conquer'.*

*Union slicing (the union of dynamic slices) is a useful technique for approximating a precise static slice. For program comprehension (and many other applications) it is often important that the union slice be an executable program, rather than merely a collection of statements which are relevant to the slicing criterion.*

*This paper presents an algorithm for computing executable union slices, using conditioned slicing. A case study is used to illustrate the algorithm and how the executable union slice is preferable to the (possibly non executable) union slice. The paper also shows, briefly, that the approach has wider applications than comprehension.*

## 1. Introduction

Understanding a large program can be a daunting task. Program slicing is useful for many applications involving program comprehension [8, 24, 47, 57], because it reduces the size of the program, making the task less daunting. Applications which require comprehension and which use slicing include performing corrective, adaptive and perfective maintenance tasks [25], debugging [35, 42] and the identification and reuse of sub-components [12, 13, 14]. All of these approaches share the observation that it is not essential to understand the behaviour and contribution of *all* of the program. Rather, it is possible, indeed preferable, to concentrate solely upon some part of the overall computation (or *aspect* to use a currently popular jargon word).

The original formulation of slicing [55] was static. That is, the slicing criterion contained no information about the input to the program. Later work on slicing created different paradigms for slicing including dynamic slicing [1, 38] (for which the input is known) and quasi-static slicing [50] (for which an input prefix is known).

Static slicing has now reached a mature stage of development. Tools, such as the Grammatech CodeSurfer system [26], can efficiently slice real-world C programs of the order of hundreds of thousands of lines of code in reasonable time [6, 7]. This paper is concerned with a generalization of slicing, called conditioned slicing[1] [11]. Tools for conditioned slicing are harder to build than static slicers, because they require symbolic execution and theorem proving in addition to static slicing. However, there exist several implementations, which are capable of producing conditioned slices for small programs [23, 19].

In this paper we show how conditioned slicing can be used to implement the union slicing of Beszedes et al. [3]. The importance of the approach advocated here is that the union slices created are *executable*[2] . This distinction between executable and non-executable slicing is an important one. For some comprehension activities it will be important to be able to execute the union slice, in order to examine the behaviour of the program for the subcomputation of interest. This is not possible with the existing approach to union slicing [3], which is non-executable. More specifically, the primary contributions of this paper are as follows:

1. The paper presents a method for constructing exe-

[1] A similar approach called constrained slicing was introduced by Field *et al.* [21].

[2] A *non-executable* slice is a subset of the program which is in fact not a slice in the semantic sense, but just a collection of statements which are relevant to the slicing criterion in some way. It may be possible to run a 'non-executable' slice, but the semantics of a such a slice will not necessarily bear any relationship to the original. The term *executable*, is misleading, but, unfortunately, it is the standard terminology in this context.

cutable union slices using conditioned slicing.

2. The paper shows how executable union slices can be useful in program comprehension (and also in test-case guided reuse and aspect re-orientation; techniques which may be deployed as an enabling technology for program comprehension).

3. The paper presents a case study which illustrates the application of an initial implementation of the executable union slicing method.

The rest of the paper is organised as follows: Section 2 gives background information on program slicing and conditioned slicing. Section 3 describes related work on the problem of union slicing. Sections 4 and 5 describe the algorithm for executable union slicing using conditioned slicing and present a detailed case study. The case study illustrates the application of our implementation of executable union slicing. Section 6 describes other applications of the approach (related to comprehension) and finally, Section 7 concludes.

## 2. Conditioned slicing

Weiser has formally defined a slice as any subset of a program that preserves a specific behaviour in respect to a criterion. The criterion, also called the *slicing criterion*, is a pair $c = (s, V)$ consisting of a statement $s$ and a subset $V$ of the variables of the analyzed program.

**Definition 1 (Weiser-style Slice)** *A slice $Slice(c)$ of a program $P$ on a slicing criterion $c$ is any executable program $P'$, where*

1. *$P'$ is obtained by deleting zero or more statements from $P$,*

2. *whenever $P$ halts on a given input $I$, $P'$ will halt for that input, and*

3. *$P'$ will compute the same values as $P$ for the variables of $V$ on input $I$.*

The most trivial (but irrelevant) slice of a program $P$ is always the program $P$ itself. Slices of interest are as small as possible, hopefully minimal.

Several different slicing techniques have been introduced to compute program slices with respect to a subset of the program executions [38, 50, 27, 11]. 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 and results in a particular program execution. For example, in dynamic slicing [38], only one program input (or test case) and the corresponding program execution are considered, while in dynamic union slicing [3] and in simultaneous dynamic slicing [38] a set of test cases are used. On the other hand in quasi static slicing [50] the set of initial states is defined by fixing the value of a subset of input variables, while the value of the remaining variables can vary. Quasi static slicing subsumes both static slicing and dynamic slicing, as this technique is able to compute static and dynamic slices depending on the fact that no variable or all variables are constrained.

Conditioned slicing [11] is a general framework for statement deletion based slicing. 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 characterize these executions is specified in terms of a first order logic formula on the input which is included in the slicing criterion, besides the program point and the set of variables of a static slicing criterion.

**Definition 2 (Conditioned Slice)** *Let $V_{in}$ be a subset of input variables of a program P, and $F_{in}$ be a first order logic formula on the variables in $V_{in}$.* A conditioned slicing criterion *of a program P is a triple $C = (F_{in}, p, V)$, where p is a statement in P and V is a subset of the variables in P.*

Canfora *et al.* [11] have demonstrated that conditioned slicing subsumes any other form of statement deletion based slicing, as the conditioned slicing criterion can be specified to obtain any form of slice. Figure 1 shows a fragment of a program which encodes the UK tax regulations in the tax year April 1998 to April 1999. A description the tax computation rules is provided in Section 4, where this program is used as a case study. Figure 2 shows the static slice with respect to the end of the program and the variable `tax`. The conditioned slice with respect to the the same criterion and the condition ($\texttt{age} \leq 60$) is shown in Figure 3.

A conditioned slice can be computed by first simplifying the program with respect to the condition on the input (i.e., discarding infeasible paths with respect to the input condition) and then computing a slice on the reduced program. A symbolic executor [17, 37] can be used to compute the reduced program, also called conditioned program in [11]. Although the identification of the infeasible paths of a conditioned program is in general an undecidable problem, in most cases implications between conditions can be automatically evaluated by a theorem prover, e.g. [10]. For example, given the condition $\texttt{age}{\leq}60$, conditioning the program in Figure 1 identifies the statements in Figure 4. This is useful because it allows the software engineer to isolate a sub-computation concerned with the initial condition of interest. The sub-program extracted can be compiled and executed as a separate code unit. It will be guaranteed to

```
personal = 4335;
pc10 = 1500;
tax = 0;
if(age >= 75)
   personal = 5980;
else if(age >= 65)
        personal = 5720;
if(age >= 65 && income > 16800)
   if(4335 > personal-((income-16800)/2))
      personal = 4335;
   else personal = personal-((income-16800)/2);
if(blind)
   personal = personal + 1380;
if(married && age >= 75)
   pc10 = 6692;
else if(married && age >= 65)
        pc10 = 6625;
else if(married || widow)
        pc10 = 3470;
if(married && age >= 65 && income > 16800)
   if (3470 > pc10-((income-16800)/2))
       pc10 = 3470;
   else pc10 = pc10-((income-16800)/2);
if(income > personal) {
   income = income - personal;
   if(income <= pc10)
      tax = (income * 10)/ 100;
   else { tax = (pc10 * 10)/100;
          income = income - pc10;
          if(income <= 28000)
 tax = tax + (income * 23)/100;
          else { tax = tax + (28000 * 23)/100;
                 income = income - 28000 ;
                 tax = tax + (income * 40)/100; }}}
code = "T";
if(!blind && !married && age < 65)
   code = "L";
else if(!blind && age < 65 && married)
        code = "H";
else if(age >= 65 && age < 75 && !married && !blind)
        code = "P";
else if(age >= 65 && age < 75 && married && !blind)
        code = "V";
```

**Figure 1. A fragment of the taxation calculation program.**

```
personal = 4335;
pc10 = 1500;
tax = 0;
if(age >= 75)
   personal = 5980;
else if(age >= 65)
        personal = 5720;
if(age >= 65 && income > 16800)
   if(4335 > personal-((income-16800)/2))
      personal = 4335;
   else personal = personal-((income-16800)/2);
if(blind)
   personal = personal + 1380;
if(married && age >= 75)
   pc10 = 6692;
else if(married && age >= 65)
        pc10 = 6625;
else if(married || widow)
        pc10 = 3470;
if(married && age >= 65 && income > 16800)
   if (3470 > pc10-((income-16800)/2))
       pc10 = 3470;
   else pc10 = pc10-((income-16800)/2);
if(income > personal) {
   income = income - personal;
   if(income <= pc10)
      tax = (income * 10)/ 100;
   else { tax = (pc10 * 10)/100;
          income = income - pc10;
          if(income <= 28000)
 tax = tax + (income * 23)/100;
          else { tax = tax + (28000 * 23)/100;
                 income = income - 28000 ;
                 tax = tax + (income * 40)/100; }}}
```

**Figure 2. Static slice on variable** `tax`**.**

```
personal = 4335;
pc10 = 1500;
tax = 0;
if(blind)
   personal = personal + 1380;
if(married || widow)
   pc10 = 3470;
if(income > personal) {
   income = income - personal;
   if(income <= pc10)
      tax = (income * 10)/ 100;
   else { tax = (pc10 * 10)/100;
          income = income - pc10;
          if(income <= 28000)
 tax = tax + (income * 23)/100;
          else { tax = tax + (28000 * 23)/100;
                 income = income - 28000 ;
                 tax = tax + (income * 40)/100; }}}
```

**Figure 3. Conditioned slice on variable** `tax` **and condition** `age` $\leq$ **60.**

```
personal = 4335;
pc10 = 1500;
tax = 0;
if(blind)
   personal = personal + 1380;
if(married || widow)
   pc10 = 3470;
if(income > personal) {
   income = income - personal;
   if(income <= pc10)
      tax = (income * 10)/ 100;
   else { tax = (pc10 * 10)/100;
          income = income - pc10;
          if(income <= 28000)
 tax = tax + (income * 23)/100;
          else { tax = tax + (28000 * 23)/100;
                 income = income - 28000 ;
                 tax = tax + (income * 40)/100; }}}
code = "T";
if(!blind && !married && age < 65)
   code = "L";
else if(!blind && age < 65 && married)
        code = "H";
```

**Figure 4. The conditioned program on condition** `age` $\leq$ **60.**

mimic the behaviour of the original if the initial condition is met.

Different automatic conditioned slicers have been implemented. The first fully automated conditioned slicer, ConSIT, was developed by Danicic *et al.* [18]. ConSIT used the Isabelle theorem prover [44]; an improved version of ConSIT is presented in [23] that exploits the Stanford Validity Checker (SVC) [2]. The SVC is more suited to proving the (many but simple and small) theorems that arise during conditioned slicing. ConSIT has also been re–implemented in WSL [52], as the system ConSUS [19]. ConSUS, allows the user to use either the Cooperating Validity Checker (CVC) [48], a version of SVC which was designed for tool-interoperability, or the built-in FermaT [51] transformation `simplify`. The `simplify` transformation is used to act as a simple (but fast) decision procedure. In this paper, the results presented in the case study were obtained using the ConSUS conditioned slicer, with the CVC theorem prover option.

## 3. Union slicing

The union of two static slices constructed for different criteria, using standard slicing algorithms will always be a slice of the union of the two criteria [3] [49, 46, 33].

By contrast, the union of two dynamic slices [38], can interfere, to alter the semantics of one of the two slices. Consider the example in Figure 5: the slice $P_1$ is constructed with respect to variable x and input $n = 1$, while slice $P_2$ is constructed with respect to variable x and input $n = 2$. The union of the two slices $P_3$ fails to preserve the semantics of the program with respect to input $n = 1$ and then cannot be considered as a valid (executable) dynamic slice for this input. Hall [27] noticed this[4] and proposed a method called simultaneous dynamic slicing to iteratively compute dynamic slices that simultaneously preserve the semantics of the original program for all the different inputs of the contributing slices.

| | $P$ | $P_1$ | $P_2$ | $P_3 = P_1 \cup P_2$ |
|---|---|---|---|---|
| 1 | read (n); | | | |
| 2 | x = 1; | x = 1; | | x = 1; |
| 3 | y = 2; | | y = 2; | y = 2; |
| 4 | if (n == 1) | | | |
| 5 | y = 1; | | | |
| 6 | if (y == 2) | | if (y == 2) | if (y == 2) |
| 7 | x = 2; | | x = 2; | x = 2; |

**Figure 5. Dynamic slices for two criteria.**

Indeed, the problem with the union of dynamic slices is that they belong to different execution traces (one for each input) and definitions along one execution trace might kill definitions along another different execution trace when simply unioning the dynamic slices. For example, in Figure 5 the definition of x at statement 7 in the union slice kills the definition of x at statement 2 on any input, while in the original program this does not happen for input $n = 1$.

It is worth noting that the problem derives from the fact that there is a dynamic dependence between statement 5 and statement 6 on the execution trace for input $n = 1$ and not on the execution trace for input $n = 2$. As statement 6 only affects the computation of x on the execution trace for input $n = 2$, neither slice $P_1$, nor slice $P_2$ include statement 5. In this way, in the union slice the data dependence between statements 5 and 6 on the execution trace for input $n = 1$ is lost, thus resulting in an erroneous result for the variable x. It is also important to remark that statement 5 transitively depends on statement 1 that defines the value of the input variable n. Such dependences are lost in current dynamic slicing algorithms whenever the dependent statement does not affect the computation of the variable of interest and then is lost in the union of the dynamic slices too.

The same problem affects other forms of slicing, such as quasi-static slicing [50] and conditioned slicing [11], where slices are constructed with respect to a subset of the execution traces. Therefore, to build valid unions for forms of slices computed with respect to subsets of execution traces,

[3] This is not true if non-dependence preserving slicing algorithms are used [41].

[4] The contribution of our example is that it is a much simpler demonstration of the problem than the example used by Hall.

we need to consider other properties than just preserving program dependences. It is likely that the union is a valid slice if the two slices are constructed with respect to the same subset of execution traces. For example, we should get valid union slices of two dependence preserving dynamic slices constructed with respect to two different variables but the same input, or of two conditioned slices constructed with respect to the same condition on the input variables. This observation is the basis for the use of conditioned slicing to compute executable union slices.

## 4. Conditioned slicing and union slices

As discussed in Section 3, the union of two slices in general does not preserve the semantics of the original program, except in the case of two static slices built using any algorithm which preserves data and control dependencies. In particular, this is true for the union of two dynamic slices. Therefore, different methods are required to build executable slices that preserve the behaviour of the original program on all the different execution traces specified by the slicing criteria, but are still smaller than the static slice. Indeed, besides the statements in the union of the slices, such a slice should include more statements to avoid the interferences of the different execution traces.

The method proposed by Hall [27] only works for dynamic slicing and not for other forms of slicing, such as quasi-static slicing and conditioned slicing that also suffer of the union problem. In particular, in the case of two conditioned slices computed with respect to the conditioned slicing criteria $C_1 = (F_1, p_1, V_1)$ and $C_2 = (F_2, p_2, V_2)$ the main problem is that the two slices would be computed on two different conditioned programs and then on two different subsets of program executions that take into account different subsets of program dependencies.

In this section we propose a method to *safely* compute executable conditioned union slices. As conditioned slicing subsumes other forms of slicing [11] the proposed method can be considered a more general method for the computation of any kind of executable union slices that preserve the behaviour of the original program with respect to the different program executions specified by the slicing criteria.

The union of two conditioned slices would be an executable slice preserving the semantics of the original program if the logic formula of the two conditioned slicing criteria are the same, i.e., $F_1 = F_2$. In this case the two slices would be computed on the same conditioned program. It is worth noting that in the case of a static slice, the conditioned program is the program itself; the formula of the slicing criterion is *true* and then the two static slices are computed with respect to the same set of program executions (all possible executions).

To compute a safe union of two conditioned slices on the two conditioned slicing criteria $C_1$ and $C_2$ above, the two conditioned slices have to be constructed with respect to the same logic formula $F$, where both $F_1 \Rightarrow F$ and $F_2 \Rightarrow F$. In this way each slice is a valid conditioned slice on the corresponding conditioned slicing criterion and the union of the two slices is still a valid slice. The formula $F$ can be constructed as the logical disjunction of the two formulas $F_1$ and $F_2$, as any formula $F$ that implies both $F_1$ and $F_2$ also implies $F_1 \vee F_2$. Below is the definition of the steps to be performed to compute a union slice based on conditioned slicing.

**Definition 3 (Conditioned Union Slicing Algorithm)** *Given a program $P$ and two conditioned slicing criteria $C_1$ and $C_2$, where $C_1 = (F_1, p_1, V_1)$ and $C_2 = (F_2, p_2, V_2)$, a conditioned union slice $S$ that preserves the behaviour of the original program on both $C_1$ and $C_2$ is achieved through the following steps:*

1. *let $F = F_1 \vee F_2$;*
2. *compute the conditioned program $P(F)$;*
3. *compute the slice $S_1$ of the conditioned program $P(F)$ with respect to the slicing criterion $(p_1, V_1)$*
4. *compute the slice $S_2$ of the conditioned program $P(F)$ with respect to the slicing criterion $(p_2, V_2)$*
5. *compute the conditioned union slice $S$ as the union of $S_1$ and $S_2$*

Consider the sample dynamic slices $P_1$ and $P_2$ in Figure 5 computed with respect to the code fragment $P$. The two dynamic slicing criteria can be expressed as two conditioned slicing criteria with respect to the two formulas $F_1 = (n = 1)$ and $F_2 = (n = 2)$. The conditioned program computed with respect to the formula $F = F_1 \vee F_2 = (n = 1 \vee n = 2)$ would include the entire code fragment $P$ and therefore the conditioned slice would correctly take into account the dependencies on both the execution traces of the two dynamic slices.

However, it is not always the case that the conditioned union slice and the static slice are the same. In the following section we will show a working example, where the conditioned union slice is smaller than the static slice.

## 5. Case Study

In this section we give examples of two dynamic slices of a program and show that the union of these two dynamic slices is not executable. It is then shown how our Conditioned Union Slicing algorithm can be used to produce a dynamic slice which does preserve the union of the two dynamic slicing criteria.

Consider the taxation calculation program shown in Figure 1. This program encodes the UK tax regulations in the tax year April 1998 to April 1999. Each person has a personal allowance which is an amount of un-taxed income. The size of this personal allowance depends upon the status of the person, which is encoded in the boolean variables `blind`, `married` and `widow`, and the integer variable `age`.

Figure 6 shows the dynamic slice with respect to the variable `tax` and input (`age` = 40, `income` = 5000, `blind` = 0, `married` = 0, `widow` = 0). Figure 7 shows the dynamic slice with respect to the variable `tax` and input (`age` = 77, `income` = 5000, `blind` = 0, `married` = 0, `widow` = 0). Basically the two dynamic criteria only differ for the value of the input variable `age`. It is worth noting that whenever `age` = 40 the variable `tax` is computed as `(income * 10)/ 100;`, while in the case `age` = 77 its value is 0. Figure 8 shows the union of the two dynamic slices. The union slice does not preserve the behaviour of the original program in the case `age` = 77. Indeed, the union slice computes the amount of `tax` as `(income * 10)/ 100` both for input variable `age` = 40 and `age` = 77.

The problem is that the union slice does not take into account the fact that whenever `age` = 77 the correct value for the variable `personal` is 5980, rather than 4335 as in the case `age` = 40. Taking this fact into account, the predicate `income > personal` in the following *if* statement is `false` and the value of `tax` does not change and remains 0. This fact is taken into account when conditioning the original program with the condition ((`age` = 40 $\vee$ `age` = 77) $\wedge$ `income` = 5000 $\wedge$ `blind` = 0 $\wedge$ `married` = 0 $\wedge$ `widow` = 0)) (steps 1 and 2 of the conditioned union slicing algorithm). The resulting conditioned program is shown in Figure 9.

By keeping in the conditioned program the possibility to change the value of the variable `personal` in case `age >= 75`, the conditioned union slice on the variable `tax` is safely computed, as shown in Figure 10 (step 3 of the conditioned union slicing algorithm). Indeed, the conditioned union slice does preserve the behaviour of the original program on both the dynamic slicing criteria and it is still much smaller than the corresponding static slice shown in Figure 2.

It is worth noting that in this particular case steps 4 and 5 of the conditioned slicing algorithm are not executed, as the variable of interest (`tax`) is the same in both the dynamic slicing criteria. In case of different variables of interest, these two steps would be needed. However, as the conditioned slices are statically computed on the conditioned program using standard data and control dependence preserving algorithms, their union would be safely computed, as discussed in Section 3.

```
personal = 4335;
pc10 = 1500;
if(income > personal) {
   income = income - personal;
   if(income <= pc10)
      tax = (income * 10)/ 100; }
```

**Figure 6. Dynamic slice on variable `tax` for input (`age` = 40 $\wedge$ `income` = 5000 $\wedge$ `blind` = 0 $\wedge$ `married` = 0 $\wedge$ `widow` = 0)).**

```
tax = 0;
```

**Figure 7. Dynamic slice on variable `tax` for input (`age` = 77 $\wedge$ `income` = 5000 $\wedge$ `blind` = 0 $\wedge$ `married` = 0 $\wedge$ `widow` = 0)).**

```
personal = 4335;
pc10 = 1500;
tax = 0;
if(income > personal) {
   income = income - personal;
   if(income <= pc10)
      tax = (income * 10)/ 100; }
```

**Figure 8. The non-executable union dynamic slice.**

```
personal = 4335;
pc10 = 1500;
tax = 0;
if(age >= 75)
   personal = 5980;
if(income > personal) {
   income = income - personal;
   if(income <= pc10)
      tax = (income * 10) / 100; }
code = "T";
if(!blind && !married && age < 65)
   code = "L";
```

**Figure 9. Conditioned program on the disjunction condition.**

```
personal = 4335;
pc10 = 1500;
tax = 0;
if(age >= 75)
   personal = 5980;
if(income > personal) {
   income = income - personal;
   if(income <= pc10)
      tax = (income * 10) / 100; }
```

**Figure 10. Conditioned union slice.**

## 6. Other Applications in Addition to Comprehension

The way in which slicing produces an executable sub-program, based upon some criterion of interest, gives rise to many applications, in addition to comprehension. For example, slicing has been applied to, among others, debugging [42, 54], testing [5, 28, 30, 31], program decomposition [25] and integration [9, 33], software metrics [4, 43, 40] and re- and reverse engineering [12, 13, 15, 16].

Similarly, having the ability to construct *executable* union slices has other applications in addition to comprehension. This section briefly describes these other applications.

### 6.1 Testing and Component Re-use

One approach to component based testing and reuse, is to use a set of test cases to act as a contract between the user of the components and their supplier. The test cases prescribe the desired behaviour of the components and can be used to support extraction of components [56, 22] and to increase confidence in their correct behaviour, with respect to the component user's requirements.

Of course, there is a potential danger here. The oft-quoted aphorism by Dijkstra [20] applies here:

> 'Testing can only show the presence of bugs; it cannot show their absence'

However, there is an approach, the X machine approach [32, 34, 29], which addresses this issue head on. In the X machine approach a set of trusted components can be integrated and the correctness of the integration process *can* be established purely by testing on a set of test cases generated from the X machine. The X machine is a form of extended Finite State Machine, with certain design for test constraints imposed to ensure that a suitable test set can be obtained.

Suppose, we use the X machine model to specify the behaviour of a system we wish to extract from a set of trusted components. The set of test data generated by the X machine could be used as a set of inputs for the formation of a set of dynamic slices. Such a set of dynamic slices could simply be unioned. However, this would not guarantee to produce an executable program. If, instead, we use the conditioned slicing approach advocated in this paper, then we would obtain an executable program which preserves the behaviour of the original upon the set of test inputs.

### 6.2 Aspect Re—Orientation

If a program is written in an aspect oriented style [36], then the effort of identifying and concentrating upon the desired aspect of behaviour may be reduced. However, if a system is not written in an aspect-oriented style, then the identification of the code which contributes to the aspect of interest may become a significant problem.

Although slicing was not designed with aspect orientation in mind, it is a technique which allows a program to be arbitrarily re-factored into sub-programs, each of which preserve the effect of the original upon some criterion of interest. As such, slicing provides a way to focus upon aspects of program's overall computation.

It is often the case that an aspect of a program's behaviour can be captured by a set of input/output pairs. This is increasingly becoming a common technique for specifying (in a contractual manner) the intended behaviour of a system. Given this approach to specification, it becomes possible to use the unioning technique to construct a union slice, where the union of dynamic criteria describe the aspect of interest. However, for such an approach to be useful, the extracted union slice must be executable.

Furthermore, even for a program written in an aspect–oriented style, it may turn out that the programmer wishes to focus upon some aspect which was not explicitly coded as an aspect by the original developer of the system. In such a situation, the original partition of the program into aspects may not be helpful; the program would require re–partitioning into aspects. Union Slicing can be useful here too, because it facilitates re-partitioning into sub-programs according to user–determined criteria.

## 7. Conclusion

It is well known that conditioned slicing subsumes both static and dynamic slicing. This paper has shown how this theoretical observation finds practical application in the construction of executable union slices, which preserve the meaning of the original program for a disjunction of dynamic slicing criteria. This work finds application, not only for program comprehension, but also for component re-use guided by software testing.

The paper illustrated, with a simple case study, that the union slice, constructed by unioning dynamic slices, may not be executable, while the executable union slice constructed using conditioned slicing is guaranteed to be executable, since it is constructed from a condition formed by a disjunction of dynamic input constraints.

## References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, New York, June 1990.

[2] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and

A. Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California, November 6–8.

[3] A. Beszedes and T. Gyimothy. Union slices for the approximation of the precise slice. In *IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 12–20, Montreal, Canada, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[4] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, Aug. 1994.

[5] D. W. Binkley. The application of program slicing to regression testing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 583–594. Elsevier, 1998.

[6] D. W. Binkley and M. Harman. An empirical study of predicate dependence levels and trends. In $25^{th}$ *IEEE International Conference and Software Engineering (ICSE 2003)*, pages 330–339, Portland, Oregon, USA, May 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

[7] D. W. Binkley and M. Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *IEEE International Conference on Software Maintenance (ICSM 2003)*, pages 44–53, Amsterdam, Netherlands, Sept. 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

[8] D. W. Binkley, M. Harman, L. R. Raszewski, and C. Smith. An empirical study of amorphous slicing as a program comprehension support tool. In $8^{th}$ *IEEE International Workshop on Program Comprehension (IWPC 2000)*, pages 161–170, Limerick, Ireland, June 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

[9] D. W. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, 1995.

[10] R. S. Boyer and J. S. Moore. Overview of a theorem-prover for A computational logic. In J. H. Siekmann, editor, *Proceedings of the 8th International Conference on Automated Deduction*, volume 230 of *LNCS*, pages 675–678, Oxford, UK, July 1986. Springer.

[11] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[12] G. Canfora, A. Cimitile, A. De Lucia, and G. A. D. Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)*, pages 424–433, Victoria, Canada, Sept. 1994. IEEE Computer Society Press, Los Alamitos, California, USA.

[13] G. Canfora, A. Cimitile, and M. Munro. $RE^2$: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance : Research and Practice*, 6(2):53–72, 1994.

[14] G. Canfora, A. De Lucia, and M. Munro. An integrated environment for reuse reengineering C code. *Journal of Systems and Software*, 42:153–164, 1998.

[15] A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: a case study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'95)*, pages 124–133, Nice, France, 1995. IEEE Computer Society Press, Los Alamitos, California, USA.

[16] A. Cimitile, A. De Lucia, and M. Munro. Qualifying reusable functions using symbolic execution. In *Proceedings of the* $2^{nd}$ *working conference on reverse engineering*, pages 178–187, Toronto, Canada, 1995. IEEE Computer Society Press, Los Alamitos, California, USA.

[17] P. D. Coward. Symbolic execution systems - a review. *Software Engineering Journal*, 3(6):229–239, Nov. 1988.

[18] S. Danicic, C. Fox, M. Harman, and R. M. Hierons. ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)*, pages 216–226, San Jose, California, USA, Oct. 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

[19] M. Daoudi, S. Danicic, J. Howroyd, M. Harman, C. Fox, L. Ouarbya, and M. Ward. ConSUS: A scalable approach to conditioned slicing. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, pages 109 – 118, Richmond, Virginia, USA, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA. Invited for special issue of the Journal of Systems and Software as best paper from WCRE 2002.

[20] E. W. Dijkstra. *A discipline of programming*. Prentice Hall, 1972.

[21] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In $22^{nd}$ *ACM Symposium on Principles of Programming Languages*, pages 379–392, San Francisco, CA, 1995.

[22] B. Fischer. Specification-based browsing of software component libraries. In *Thirteenth International Conference on Automated Software Engineering*, pages 74–83. IEEE Computer Society Press, 1998.

[23] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. ConSIT: a fully automated conditioned program slicer. *Software—Practice and Experience*, 34:15–46, 2004. Published online 26th November 2003.

[24] C. Fox, M. Harman, R. M. Hierons, and S. Danicic. Backward conditioning: a new program specialisation technique and its application to program comprehension. In $9^{th}$ *IEEE International Workshop on Program Comprenhesion (IWPC'01)*, pages 89–97, Toronto, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

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

[26] Grammatech Inc. The codesurfer slicing system, 2002.

[27] R. J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, Mar. 1995.

[28] M. Harman and S. Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3):143–162, Sept. 1995.

[29] R. M. Hierons and M. Harman. Testing against non-deterministic stream X-machines. *Formal Aspects of Computing*, 12:423–442, 2000.

[30] R. M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[31] R. M. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability*, 12:23–28, Mar. 2002.

[32] M. Holcombe and F. Ipate. *Correct systems — Building and business process solution*. Springer, 1998.

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

[34] F. Ipate and M. Holcombe. Generating test sequences from non-deterministic X-machines. *Formal Aspects of Computing*, to appear.

[35] M. Kamkar. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993. Available as Linköping Studies in Science and Technology, Dissertations, Number 297.

[36] G. Kiczales. Aspect oriented programming. *ACM SIGPLAN Notices*, 32(10):162–162, Oct. 1997. No paper in volume, but table of contents includes an entry for this invited talk.

[37] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[38] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[39] H. D. Longworth, L. M. Ott, and M. R. Smith. The relationship between program complexity and slice complexity during debugging tasks. In *Proceedings of the Computer Software and Applications Conference (COMPSAC'86)*, pages 383–389, 1986.

[40] A. D. Lucia, M. Harman, R. M. Hierons, and J. Krinke. Union of slices are not slices. In $7^{th}$ *European Conference on Software Maintenance and Reengineering*, pages 363–367, Benevento, Italy, Mar. 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

[41] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In $2^{nd}$ *International Conference on Computers and Applications*, pages 877–882, Peking, 1987. IEEE Computer Society Press, Los Alamitos, California, USA.

[42] L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the* $11^{th}$ *ACM conference on Software Engineering*, pages 198–204, May 1989.

[43] L. C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828:xvii + 321, 1994.

[44] T. Reps and W. Yang. The semantics of program slicing. Technical Report Technical Report 777, University of Wisconsin, 1988.

[45] J. Rilling, A.Seffah, and J.Lukas. MOOSE – a software comprehension framework. In $5^{th}$ *World Multi-Conference on systemics, cybernetics and informatics (SCI 2001)*. to appear.

[46] A. Stump, C. W. Barrett, and D. L. Dill. CVC: a cooperating validity checker. In J. C. Godskesen, editor, *Proceedings of the International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, 2002.

[47] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[48] G. A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada, June 1991. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[49] M. Ward. Assembler to C migration using the FermaT transformation system. In *IEEE International Conference on Software Maintenance (ICSM'99)*, Oxford, UK, Aug. 1999. IEEE Computer Society Press, Los Alamitos, California, USA.

[50] M. Ward, F. W. Calliss, and M. Munro. The maintainer's assistant. In *Proceedings of the International Conference on Software Maintenance 1989*, page 307. IEEE Computer Society Press, Los Alamitos, California, USA, 1989.

[51] M. Weiser. Programmers use slicing when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[52] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[53] N. Wilde, J. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 200–205, Orlando, Florida, U.S.A., 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[54] J. Zhao. Slicing aspect-oriented software. In $10^{th}$ *IEEE International Workshop on Program Comprehension*, pages 351–260, Paris, France, June 2002. IEEE Computer Society Press, Los Alamitos, California, USA.