Backward Conditioning: a new program specialisation technique and its application to program comprehension

Chris Fox	Mark Harman and Rob Hierons	Sebastian Danicic
King's College	Brunel University	Goldsmiths College
University of London	Uxbridge	University of London
Strand	Middlesex	New Cross
London WC2R 2LS	UB8 3PH	London SE14 6NW
United Kingdom	United Kingdom	United Kingdom
Tel: +44 (0)20 7848 2694	Tel: +44 (0)1895 274 000	Tel: +44 (0)20 7919 7868
Fax: +44 (0)20 7848 2851	Fax: +44 (0)1895 251 686	Fax: +44 (0)20 7919 7853
foxcj@dcs.kcl.ac.uk	mark.harman@brunel.ac.uk	sebastian@mcs.gold.ac.uk

Keywords: Conditioned program slicing, program specialisation, path condition analysis

Abstract

This paper introduces backward conditioning. Like forward conditioning (used in conditioned slicing), backward conditioning consists of specialising a program with respect to a condition inserted into the program.

However, whereas forward conditioning deletes statements which are not executed when the initial state satisfies the condition, backward conditioning deletes statements which cannot cause execution to enter a state which satisfies the condition. The relationship between backward and forward conditioning is reminiscent of the relationship between backward and forward slicing.

Forward conditioning addresses program comprehension questions of the form 'what happens if the program starts in a state satisfying condition c?', whereas backward conditioning addresses questions of the form 'what parts of the program could potentially lead to the program arriving in a state satisfying condition c?'.

The paper illustrates the use of backward conditioning as a program comprehension assistant and presents an algorithm for constructing backward conditioned programs.

1. Introduction

Program comprehension often starts with a programmer inspecting source code, asking questions such as

> "What happens when the initial value of `Balance` is less than zero?"

and

> "How could this program get into a state where `temperature` is greater than `100` at this point?"

or constructing hypotheses such as

> "This program could never get to this point and have the `file_lock` flag set to `true`."

These questions, and their answers, are important aspects of the comprehension activity.

Previous work on conditioned slicing [4, 5, 7] has considered the way in which such questions can be investigated. This work helps to answer comprehension questions which solely concern propagation of state information in a *forward direction* from *initial* states. Conditioned slicing would assist in answering questions like the first of the three above, but not the second two.

Unfortunately, many questions concern *intermediate* and *final* states in which information needs to be propagated *backwards* from the condition. Traditional conditioned slicing is a forward-propagation technique. This paper introduces a counterpart to this traditional conditioning, termed backward conditioning. Hereinafter, the condition used in the traditional approach to conditioned slicing will be referred to as a 'forward condition' and a condition that is used to eliminate program code that proceeds will be refered to as a 'backward condition'.

Constructing a slice with respect to a backward condition, p consists of removing statements which cannot lead the program into a state which satisfies p. The code which remains is the slice. It contains code which potentially could lead the program into a state which satisfies p.

To illustrate consider the simple example fragment in Figure 1. The program is an idealized fragment of code concerned with bank account management. The programmer might be interested to see which parts of the program could finish in a state where the account balance (`bal`) was negative. That is, the condition of interest is

```
bal < 0
```

and the point of interest is the end of the fragment. Constructing a slice with respect to the backward condition yields the slice depicted in the right-hand column of Figure 1. In this case, backward conditioning has removed all but one assignment to `bal`, indicating that only this remaining assignment can lead the program into a state where `bal` is negative. In this way, backward conditioning assists the programmer by focusing attention on the statements which can potentially cause a situation of interest to arise within the program.

A conditioned slice can be constructed with respect to a mixture of of conditions applied in the forward or backward direction. There is therefore a need for some convenient notation to denote a general conditioned slicing criterion. A condition that is to be applied in the forward direction, p, will be denoted by a downward pointing subscript arrow before the condition, contained within "ceiling" brackets, thus ${}_{\downarrow}\lceil p \rceil$. This distinguishes it from a condition p to be applied in the backward direction, which will be denoted with an upward pointing superscript arrow, with the condition contained within "floor" brackets, thus ${}^{\uparrow}\lfloor p \rfloor$. The arrows indicate the direction in which the condition is to be appled with respect to the program text, and the (optional) distinctive brackets aid disambiguation of the scope of the arrows in complex conditions. Furthermore, a condition (either forward or backward) can be inserted anywhere within the program (not just at the beginning of the code as with traditional conditioned slicing). Finally, there is no need to restrict oneself to a single condition. Therefore, the conditioned slicing criterion is generalised to a set of pairs. Each pair contains either

- A traditional 'static' criterion: a set of variables and point of interest or
- A condition/program point pair

In their most general form the static part of the criteria also has a direction, following the introduction of forward static slicing [9], which mirrors backward slicing in much the same way as backward conditions mirror forward conditions. However, the focus of this paper is the introduction of backward conditions and their use in program comprehension, so this possibility will not be explored further in the present paper.

Using this generalised notation, the conditioned slicing criterion for the bank account program in Figure 1 would be denoted

$$\{({}^{\uparrow}\lfloor \texttt{bal < 0} \rfloor, 9), (\{\texttt{bal}\}, 9)\}$$

The rest of this paper is organised as follows. Section 2 presents a simple case study to show how backward conditioning can be used in a program comprehension setting. Section 3 presents an algorithm for computing backward conditioned programs, based upon an augmentation of the ConSIT approach [5]. Section 4 describes the relationship between traditional slicing and backward and forward conditioning and Section 5 concludes.

2. Application to Program Comprehension

This section briefly illustrates the way backward conditions can be used to assist in program comprehension. Consider the simple tax calculation program in Figure 2, which was used in [5] to illustrate the use of forward conditions, implemented by the ConSIT system.

The program represents a computation of tax codes and amounts of tax payable, including allowances for a United Kingdom citizen in the tax year April 1998 to April 1999. Each person has a personal allowance which is an amount of un-taxed income. The personal allowance depends upon the status of the person, reflected by the boolean variables `blind`, `married` and `widowed` and the integer variable `age`. There are three tax bands, for which tax is charged at the rates of 10%, 23% and 40%. The width of the 10% tax band is subject to the status of the person, while the 23% and 40% are fixed for all individuals. This set of taxation rules constitutes a governmental 'business system', and the program in Figure 2 represents an attempt to capture these rules in program code.

While forward conditions are helpful in understanding this program, backward conditions provide a useful additional tool in the armory of program comprehension.

For example, suppose that the programmer is interested in the personal allowance. The maximum personal allowance that any individual could obtain under the UK taxation rules in 1999 was £7360. Who was entitled to this allowance? Using backward conditioning the condition

```
personal = 7360
```

can be inserted at the end of the program. The variable of interest in this case is the final value of `personal`. Therefore the conditioned slicing criterion is

$$\{({}^{\uparrow}\lfloor \texttt{personal = 7360} \rfloor, \mathit{end}), (\{\texttt{personal}\}, \mathit{end})\}$$

`1 if (bal>0)`	`1 if (bal>0)`
`2   if (bal>wdraw)`	`2   if (bal>wdraw)`
`3     bal = bal - wdraw`	
`    else`	`    else`
`4   { gap = bal - wdraw;`	`4   { gap = bal - wdraw;`
`5      if (gold(cust) && gap < top)`	`5      if (gold(cust) && gap < top)`
`6        { pinkzone = wdraw - bal;`	
`7          bal = 0; }`	
`8      else bal = bal - wdraw ;`	`8      else bal = bal - wdraw ;`
`    }`	`    }`
`9  printf("%d",bal) ;`	`9  printf("%d",bal) ;`
⋮	⋮
Original	Result of backward conditioning

Figure 1. Backward conditioning on `bal < 0` at line 9

Slicing with respect to this criterion yields the slice in Figure 3. This program contains computations which could lead to the final value of the variable `personal` being £7360. It removes all computation which cannot leave the program in this state. Therefore, the programmer can conclude that the program will not award a personal allowance of £7360 unless the individual is blind and at least 75 years of age.

In this way backward conditioning allows speculative hypotheses about the program's behaviour to be investigated. It does not, in general, answer these questions completely, because the programmer still has some program code to consider. However, it assists the human by automatically removing portions of code which are not relevant to the question under consideration.

Occasionally, the programmer will phrase a question about the execution as a backward condition, and will receive a definitive answer. This happens when the slice is empty, revealing that the backward condition can never arise. This is helpful in asking questions which serve as 'sanity checks'. For example, in the case of taxation, no individual should (under the 1999 UK law) receive an overall income tax burden of 40% or more. This can be checked by appending to the end of the program backward condition,

$$\texttt{(tax >= income}_0\texttt{ * 40)/100}$$

where $\texttt{income}_0$ captures the original value of the variable `income` (i.e. the individual's gross income). This condition asserts that the amount of tax paid is at least 40% of the individual's gross income. Since this is not possible, conditioning the program with respect to this backward condition yields the empty program. More formally, the empty slice is obtained for the slicing criterion

$$\{(^{\uparrow}\lfloor\texttt{(tax >= income}_0\texttt{ * 40)/100}\rfloor, end), \\ (\{\texttt{tax}\}, end)\}$$

3. Computing Backward Conditioned Slices

As with conventional forward slicing [4, 5], automated backward conditioning requires symbolic execution together with automatic theorem proving. To eliminate irrelevant paths, for each statement (or statement block) the algorithm has to determine whether all paths through that statement lead to the negation of the required condition. This is achieved by: (i) extending the notion symbolic execution so that a path which leads to the negation of the required condition(s) are given a distinguished value $\perp$; (ii) for each statement determining whether all paths lead to $\perp$, in which case the statement is irrelevant for the purpose of obtaining the required condition.

Theorem proving is computationally expensive. To reduce the number of theorems to be proved during backward conditioning, a practical system might slice the program first, perhaps using a default slicing criterion consisting of the variables mentioned in the required backwards condition.

Conceptually, the system is thus comprised of the following components:

1. Static Slicer
2. Symbolic Executor
3. Theorem Prover

The slicer first eliminates statements on program dependence grounds. The symbolic executor and theorem prover essentially seek to eliminate additional statements on path-condition grounds: the symbolic executor provides a set of path-state pairs $\{\langle\pi_1,\sigma_1\rangle,\ldots,\langle\pi_n,\sigma_n\rangle\}$ for each statement [5], and this information is used by the theorem prover to determine the relevant paths and statements. To simplify

```
main() {
int age, blind, widow, married, income;
int personal, tax, t;

scanf("%d",&age);
scanf("%d",&blind);
scanf("%d",&married);
scanf("%d",&widow);
scanf("%d",&income);

if (age>=75) personal = 5980;
else if (age>=65) personal = 5720;
else personal = 4335;

if ((age>=65) && income>16800) {
  t = personal - ((income-16800)/2);
  if (t>4335) personal = t;
  else personal = 4335;}

if (blind) personal = personal + 1380;

if (married && age>=75) pc10 = 6692;
else if (married && (age>=65)) pc10 = 6625;
else if (married || widow) pc10 = 3470;
else pc10 = 1500;
```

```
if (married && age>=65 && income>16800) {
  t = pc10-((income-16800)/2);
  if (t>3470) pc10 = t; else pc10 = 3470;}

if (income<=personal) tax = 0;
else {
  income = income-personal;
  if (income<=pc10) tax = income/10;
  else {
    tax = pc10/10;
    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;}}}

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’;
else code = ’T’;
}
```

Figure 2. UK Income Taxation Calculation Program

the implementation, we can introduce the backward condition p at line l, $(^{\uparrow}\lfloor p \rfloor, l)$, by way of a new program statement `assertb(p)` that is inserted at line l.

In order to determine whether a statement block s can contribute to obtaining the desired condition c, we need to determine all the execution paths $\pi_1, \ldots, \pi_n$ that pass through s, together with their related symbolic states $\sigma_1, \ldots, \sigma_n$. If every path π_k $(1 \leq k \leq n)$ through that statement block can be shown to lead to the negation of c, when evaluated in the corresponding state σ_k, then that block can be coloured as being irrelevant for obtaining the desired condition.

The algorithm for backward condition path reduction has the steps given below:

1. Label statements.
2. Find all execution paths and symbolic states.
3. Associate each path with the labels of the statements on that path.
4. Eliminate every statement which is not associated with any path that can support the required condition.

The detail of each of these steps is as follows:

1. Each statement (or statement block) is associated with a unique identifier (e.g. a line number).[1]

2. The symbolic executor finds all of the possible symbolic execution paths through the program. The leaves of the execution "tree" for a conventional program are denoted by pairs of path conditions and symbolic states. The behaviour of the symbolic executor is extended in such a way that paths where the required condition(s) are not obtained have the distinguished value $\perp$ as their symbolic path-state.[2]

3. The leaves of the execution tree each correspond with the symbolic result of a possible path of execution. We associate each leaf of the execution tree with the set of statement identifiers that denote the statements that are executed on the corresponding path.

[1] For the sake of simplicity, we talk of eliminating individual statements in this algorithm. However, it is generally better to consider eliminating statement blocks rather than individual statements, as this reduces the time complexity of the algorithm: the number of calls to the theorem prover is then determined by the number of symbolic execution paths, rather than the product of the number of statements and paths.

[2] With a single backward condition, only statements leading up to the condition need be considered. Also, for the purpose of symbolic execution, different numbers of non-zero iterations of a terminating loop for can be considered as equivalent.

```
main() {
int age, blind;
int personal;

 scanf("%d",&age);
 scanf("%d",&blind);
 if (age>=75) personal = 5980;
 if (blind) personal = personal + 1380 ;
}
```

Figure 3. Highest Personal Allowance

4. If all the paths through a statement (i.e. all of the paths which are associated with that statement's identifier) are $\perp$ (that is, they can be shown not to support the required condition) then that statement can be eliminated. This has the effect of eliminating code that is on paths that can be shown to lead to the negation of the required condition.

This implementation requires the incorporation of a theorem prover into the symbolic executor. Currently, our implementation work is focused on the Stanford Validity Checker (SVC), which is able to reason effectively with arithmetic expressions and linear inequalities [3, 2], as well as the Isabelle, which is a more general purpose theorem prover [14, 15, 16]. Both SVC and Isabelle have been successfully integrated with a Prolog-based symbolic executor and Java-based slicer (Espresso) [6] to achieve forward conditioned program slicing [5]. Work is underway to extend the implementation to incorporate the backward conditioning algorithm presented here.

3.1. Example of Backward Conditioning

Figure 4 is a simple example of a program containing a conditional which we will backward condition. Conditional expressions, such as those declared by `assertb`, have to be interpreted in the relevant (symbolic) state. A condition c occurring in the context generated by program p is evaluated in each of the symbolic states that arise on the various paths through p. If the expression $\pi \triangleright \sigma$ is one of the path-state pairs generated by symbolic execution of p—where π is the path condition, and σ is the corresponding symbolic state—then c is to be evaluated in symbolic state σ, which we write $I_\sigma(c)$, treating σ as a partial function from variables to their symbolic values. Essentially this gives the interpretation of the program statement c in the state σ, as in ordinary state-based program semantics, except that here the state σ is a symbolic state.

In order to perform backward conditioning, it is necessary to check, for each statement, whether there are any paths through that statement that might allow us to obtain the required condition. If all paths through a statement allow us to demonstrate that the negation of the condition holds, then that statement can be deleted.

Figure 5 plots key statements against the paths upon which they appear. For path-state (i), namely $(a_0 > 10) \triangleright (\mathtt{a} = 20a_0, \mathtt{a_0} = a_0)$, to determine whether that path could lead to the satisfaction of the required condition, the question is whether:

$$a_0 > 10 \vdash \neg I_{(\mathtt{a}=20a_0,\mathtt{a_0}=a_0)}(\mathtt{a} < \mathtt{a_0})$$

where the sequent $a \vdash b$ means that we can derive b given a. In this case, b is the negation of the condition $\mathtt{a} < \mathtt{a_0}$ asserted by the statement $\mathtt{assertb(a < a_0)}$ when evaluated in the appropriate symbolic state. If the sequent can be shown to be valid, then the path does not give rise to the required condition. If all paths through a given statement block do not give rise to the required condition, then the statements in that block can be elided.

So, evaluating the backward condition in the final symbolic state on this path gives us:

$$a_0 > 10 \vdash \neg(20a_0 < a_0)$$

which is **valid**.

For path (ii) we have the following:

$$a_0 \not> 10 \vdash \neg I_{(\mathtt{a}=0,\mathtt{a_0}=a_0)}(\mathtt{a} < \mathtt{a_0})$$

Performing the substitution, this gives us:

$$a_0 \not> 10 \vdash \neg(0 < a_0)$$

which is not **valid**.

Summarising these proofs we have the following, for path (i):

1. $a_0 > 10 \vdash \neg I_{(\mathtt{a}=20a_0,\mathtt{a_0}=a_0)}(\mathtt{a} < \mathtt{a_0})$
2. $a_0 > 10 \vdash \neg(20a_0 < a_0)$
3. **valid**

and for path (ii):

program code	*comment*
`main() {`	
`int a;`	
`scanf{"%d", &a};`	This statement occurs on path (i) and (ii) below. At this point, the path condition is empty, and the symbolic state is $\mathtt{a} = \mathtt{a}_0$, where a_0 is a unique symbolic constant, standing for the unknown input value. We write this as $\top \triangleright \mathtt{a} = a_0$.
`a0 = a;`	Added to capture initial value of `a`. Path condition and symbolic state: $\top \triangleright \mathtt{a} = a_0, \mathtt{a}_0 = a_0$
`if (a > 10) {`	
`a = a * 20;`	This statement, call it A, only occurs on path (i), below. Path condition and symbolic state: $a_0 > 10 \triangleright \mathtt{a} = 20a_0, \mathtt{a}_0 = a_0$
`} else {a = 0;`	This statement, call it B, occurs on path (ii), below. Path condition and symbolic state: $a_0 \not> 10 \triangleright \mathtt{a} = 0, \mathtt{a}_0 = a_0$
`}`	Path condition and symbolic states at this point: (i) $a_0 > 10 \triangleright \mathtt{a} = 20a_0, \mathtt{a}_0 = a_0$ (ii) $a_0 \not> 10 \triangleright \mathtt{a} = 0, \mathtt{a}_0 = a_0$
`assertb(a < a0);`	This statement has been added to state the condition of interest. In essence, it indicates that we are interested in determining which parts of the program are responsible for decreasing the value of variable `a` compared to its original input value, as recorded by $\mathtt{a}_0$
`}`	

Figure 4. A Simple Program and its Paths

1. $a_0 \not> 10 \vdash \neg I_{(\mathtt{a}=0,\mathtt{a}_0=a_0)}(\mathtt{a} < \mathtt{a}_0)$
2. $a_0 \not> 10 \vdash \neg(0 < a_0)$
3. **not valid**

So executions along path (i), through statement 'A' (`a = a * 20`), definitely cannot allow the asserted condition to be true, but executions following path (ii), through statement 'B' (`a = 0`), might. As no computations along path (i) can satisfy the asserted condition, statements that are exclusive to that path can be eliminated.

What if the condition had been `assertb(a > a0)`? For path (i) we would have the following derivation:

1. $a_0 > 10 \vdash \neg I_{(\mathtt{a}=20a_0,\mathtt{a}_0=a_0)}(\mathtt{a} > \mathtt{a}_0)$
2. $a_0 > 10 \vdash \neg(20a_0 > a_0)$
3. **not valid**

For path (ii) we would have:

1. $a_0 \not> 10 \vdash \neg I_{(\mathtt{a}=0,\mathtt{a}_0=a_0)}(\mathtt{a} < \mathtt{a}_0)$
2. $a_0 \not> 10 \vdash \neg(0 < a_0)$
3. **not valid**

¿ From this we can see that execution paths through statement 'A' make the condition true, and although execution paths through 'B' do not necessarily lead to the truth of the condition, they include computations in which the condition is true. Thus neither statement can be eliminated.

4. Related Work

Program slicing was introduced by Weiser [17, 19]. The original intention of slicing was to assist in program debugging [13], but Weiser also empirically investigated slicing's ability to assist program comprehension [18]. Weiser's slicing criterion consisted of a set of variables of interest V and a point of interest n within the original program. Statements which cannot affect the values of variables in V at n are removed to form the slice.

For example, consider the program in section (a) of Figure 6. Slicing this program with respect to the criterion $(\{\mathtt{x}\}, 8)$ yields the slice in section (b). Slicing in this way assist program comprehension by removing the computation on `y`, allowing the programmer to focus on the computation on `x`.

Korel and Laski [11] introduced dynamic slicing as a counterpart to Weiser's original static formulation, and more recently investigated the use of dynamic slicing in program comprehension [12]. The dynamic slicing criterion

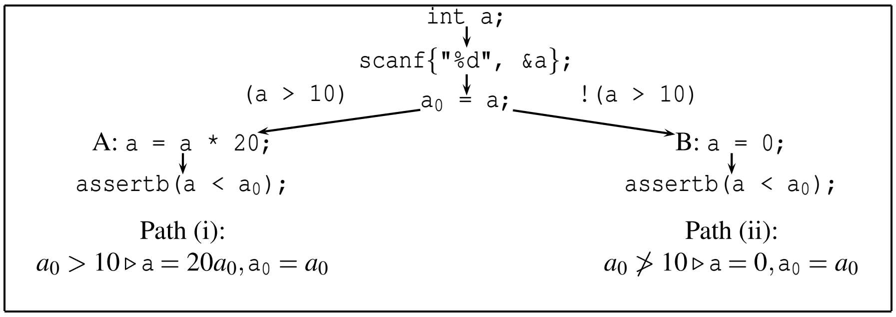

Figure 5. Execution Paths Made Explicit

augments the static criterion with a sequence of input values. The dynamic criterion is thus a triple, (I,V,n), where I is an input sequence and (V,n) is the static slicing criterion.

Conditioned slicing [4, 5, 7] augments Weiser's traditional static slicing criterion with a condition which captures a set of initial program states of interest. This allows a programmer to further specialize a program by eliminating statements which do not contribute to the computation of the variables of interest when the program is executed in one of the initial states of interest.[3]

The conditioned slicing criterion is thus a triple, (p,V,n) where p is some initial condition of interest and (V,n) are the two components of the 'static' slicing criterion. For example, the conditioned slice of the original program in section (a) of Figure 6 for the criterion $(\mathtt{x>0}, \{\mathtt{x}\}, 8)$ is shown in section (c) of the figure. This slice is also the dynamic slice for all input sequences in which the first element, x of the sequence satisfies the condition $x > 0$.

Conditioned slicing is more useful for comprehension than either static or dynamic slices because it subsumes and generalizes both [4]. However, as has been demonstrated in the present paper, there are situations where it is useful to generalise the conditions used in conditioning a program, allowing both backward and forward conditions. This generalization requires a notation which allows slicing criteria to contain an arbitrary number of both forward and backward conditions. Using this generalised notation, the slicing criterion for the traditional conditioned slice in section (c) of Figure 6 is reformulated as

$$\{(_{\downarrow}\lceil \mathtt{x>0} \rceil, 1), (\{\mathtt{x}\}, 8)\}$$

Forward conditioning assists the programmer by considering the effect of propagating state information forward from a condition. This addresses questions of the form

> "what would happen if the program continued from here in some state satisfying p."

[3] Program conditioning exploits a form of feasible path analysis. This kind of analysis has also been used in the context of program testing [8, 10].

However, the effect of backward conditioning is the mirror-image of that for forward conditioning. For example, section (d) of Figure 6 shows the effect of backward conditioning the original program in section (a) with respect to the condition $\{(^{\uparrow}\lfloor \mathtt{x>0} \rfloor), 8), (\{\mathtt{x}\}, 8)\}$. This slice removes code which cannot leave the program in a final state satisfying `x>0`.

Backward conditioning assists the programmer by considering the effect of propagating state information backward from a condition. This addresses questions of the form

> "how could the program have arrived here in some state satisfying p."

Of course, the generalisation presented here allows for both forms of condition to be combined. For example, consider the conditioned slicing criterion $\{(^{\uparrow}\lfloor \mathtt{x<=0} \rfloor), 8), (_{\downarrow}\lceil \mathtt{x>0} \rceil, 1), (\{\mathtt{x}\}, 8)\}$. For this criterion the slice is the empty program, revealing that it is impossible for the program in Figure 6 to start off with `x` being positive and finish up with it being negative.

In the example of Figure 4, rather than delete the program statement, it might be more informative to colour it (or display it 'greyed out') [1]. If this idea of colouring statements is adopted, then the backward condition can be more informative even in the subsequent example where the condition is changed to $\mathtt{a} > \mathtt{a}_0$: colours can be used to indicate that when a particular statement is executed, the backward condition will always be true—the then part of the conditional in the second example—and when the status of the backward condition is contingent—as with the else part of the conditional in both of these examples.

It may be appropriate to aim to remove irrelevant statements from *within* a path, rather than merely eliminate irrelevant paths. One way this can be achieved is by using counter-factual reasoning. For each statement we symbolically execute the program with that statement removed, and see whether we can show that the variables of interest will have the same value compared with the original program. If so, then that statement may safely be removed. Essentially

`1 scanf("%d",&x);`
`2 y=2*x;`
`3 if (y>x)`
`4   { x=x+1;`
`5     y=y*y;}`
`  else`
`6   { x=x*2;`
`7     y=y-x;}`
`8 printf("%d",x);`
	`1 scanf("%d",&x);`
	`2 y=2*x;`
	`3 if (y>x)`
	`4     x=x+1;`
	
	`  else`
	`6     x=x*2;`
		`1 scanf("%d",&x);`
		`2 y=2*x;`
		`3 if (y>x)`
		`4     x=x+1;`
			`1 scanf("%d",&x);`
			`2 y=2*x;`
			`3 if (y>x)`
			
			`  else`
			`6     x=x*2;`

(a) Original	(b) slice on $(\{x\},8)$	(c) Conditioned on ${}_{\downarrow}\lceil \texttt{x>0} \rceil$	(d) Conditioned on ${}^{\uparrow}\lfloor \texttt{x>0} \rfloor$

Figure 6. Comparison of forward and backward conditioning

this is semantic slicing. This is a very expensive analysis and in most cases standard static slicing will achieve results that are almost as good much more cheaply.

5. Conclusions and Future Work

This paper introduces a new program specialisation technique call 'backward conditioning', a counterpart to forward conditioning, used in conditioned program slicing.

It is argued that backward conditioning is useful as a supporting technology in program comprehension. It allows a programmer to explore the answer to questions such as

> "How could the program reach this point with a negative value for x?"

and

> "Which statements could make x equal y at this point?"

Backward conditioning propagates state information backward from the condition point to delete statements which could not cause execution to satisfy the condition, whereas forward conditioning propagates state information forward from the condition to delete statements which could not be executed were the condition to be satisfied. In this way backward conditioning provides a complement to traditional forward conditioning, which allows questions like

> "Which statements would have been executed were x to equal y at this point?"

or

> "What happens if execution continues from this point with a negative value for x?"

It would be interesting to combine backward and forward conditioning and backward and forward slicing into a single unified program analysis technique. This would potentially create a powerful and highly general technique for program specialisation, which would allow the programmer to explore the answer to sophisticated questions about program behaviour as a part of program comprehension activity. This combination of backward and forward conditioning remains a problem for future work.

References

[1] T. Ball and S. G. Eick. Visualizing program slices. In A. L. Ambler and T. D. Kimura, editors, *Proceedings of the Symposium on Visual Languages*, pages 288–295, Los Alamitos, CA, USA, Oct. 1994. IEEE Computer Society Press.

[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] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Design Automation Conference*, June 1998. San Francisco, CA.

[4] 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.

[5] 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.

[6] S. Danicic and M. Harman. Espresso: A slicer generator. In *ACM Symposium on Applied Computing, (SAC'00)*, page To appear, Como, Italy, Mar. 2000.

[7] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, pages 9–18, Berlin, Germany, Mar. 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[8] A. Goldberg, T. C. Wang, and D. Zimmerman. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 international symposium on Software testing and analysis*, pages 80–94, Seattle, WA USA, August 1994.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[10] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *Proceedings of the 1994 international symposium on Software testing and analysis*, pages 95–107, Seattle, WA USA, August 1994.

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

[12] B. Korel and J. Rilling. Dynamic program slicing in understanding of program execution. In 5^{th} *IEEE International Workshop on Program Comprehesion (IWPC'97)*, pages 80–89, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[13] 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.

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

[15] L. C. Paulson. Isabelle's reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1997.

[16] L. C. Paulson. Strategic principles in the design of Isabelle. In *CADE-15 Workshop on Strategies in Automated Deduction*, pages 11–17, Lindau, Germany, 1998.

[17] M. Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

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

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