Program Simplification as a Means of Approximating Undecidable Propositions

Mark Harman,
Chris Fox,
Robert Hierons,
Goldsmiths College,
University Of London,
New Cross,
London SE14 6NW, UK.
Tel: +44 (0)171 919 7860
Fax: +44 (0)171 919 7853
M.Harman@gold.ac.uk

David Binkley,
Computer Science Department,
Loyola College in Maryland,
4501 North Charles Street,
Baltimore, Maryland 21210, USA.
Tel: +1 410 617 2881
Fax: +1 410 617 2157
binkley@cs.loyola.edu

Sebastian Danicic,
School of Informatics,
University of North London,
Eden Grove,
London, N7 8DB, UK.
Tel: +44 (0)171 607 2789
Fax: +44 (0)171 753 7009
S.Danicic@unl.ac.uk

Keywords: Testing, Slicing, Transformation

Abstract

We describe an approach which mixes testing, slicing, transformation and formal verification to investigate speculative hypotheses concerning a program, formulated during program comprehension activity.

Our philosophy is that such hypotheses (which are typically undecidable) can, in some sense, be 'answered' by a partly automated system which returns neither 'true' nor 'false', but a program (the 'test program') which computes the answer.

The motivation for this philosophy is the way in which, as we demonstrate, static analysis and manipulation technology can be applied to ensure that the resulting test program is significantly simpler than the original program, thereby simplifying the process of investigating the original hypothesis.

1 Introduction

Often the task of legacy system, source code comprehension begins with a series of questions about properties for the system. These questions represent speculative hypotheses about the supposed behaviour of the system.

Ideally it would be attractive to have a fully automated formal reasoning tool, which would be sufficiently powerful to decide these propositions for us. With such a system we would be able to insert pre- and post- conditions into a legacy system to capture our hypotheses and then ask the tool to decide whether the post-condition was implied by the precondition. Clearly, this would be of considerable assistance in comprehension effort.

Unfortunately, because all but the most trivial propositions will concern properties of the system which are undecidable, such a quixotic tool is impossible. In its place, testing is often relied upon as the mechanism to investigate hypotheses.

By recasting the problem of answering such questions as a simplification problem, we circumvent undecidability, because the simplification process *approximates* the answer. This gives us a way of partially automating support for comprehension, based on the formulation and investigation of hypotheses.

Our approach is reminiscent of Gaudel's approach to testing [5], in which the correctness of a system is established in part by testing and in part by proof. Or perhaps, that the part of the system's correctness which remains unestablished by test cases, is formalised as a hypothesis. A similar approach is adopted Holcombe and Ipate [8], where the primitive components of a system are assumed to be correct with test effort guaranteeing that, if these components *are* correct and their integration satisfies a few natural constraints, then a well defined finite set of test cases can be used to determine the correctness of the system as a whole.

Our concern is not correctness but comprehension, where the human is inevitably involved, but the machine should be able to automate tiresome repetitious, rule-based tasks required to facilitate comprehension. The situation is illustrated in Figure 1. The shaded region represents human comprehension effort, the unshaded region represents automated sim-

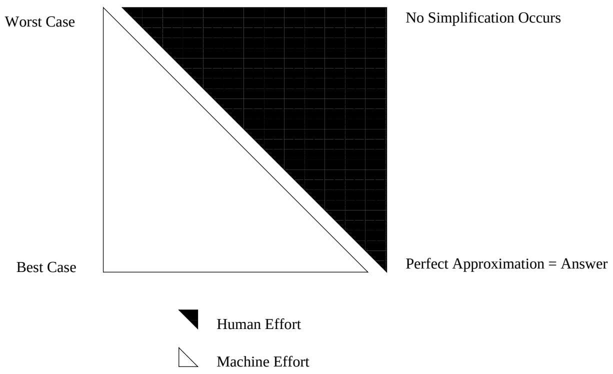

Figure 1: Yin–Yang: Human Computer Interface

plification effort. In the best case the system produces the simplest possible program, immediately denoting the answer. In the worst case no simplification is possible. An approach which attempts to *decide* the question, effectively collapses this diagram into one of these two possible scenarios. Using program simplification as approximation we open up the entire spectrum: In general *some* simplification will occur, reducing the human effort required to answer the question, but not eliminating it. The extreme cases are preserved, so nothing is lost from the old bi-polar world of definite answers. In the work of Holcombe, Ipate and Gaudel, the shaded region represents human proof effort and the unshaded region automated test generation. The difference is that in their work on testing, Holcombe, Ipate and Gaudel are concerned with the precise specification of the boundary between the white and black sections of Figure 1, whereas our concern is to improve algorithms driving them further down the diagram towards the best case.

The rest of this paper is organized as follows:

In section 2 we introduce a theory of 'programs as answers' and in section 3 we show how 'test programs' can be simplified. Section 4 argues that the simplified test programs are partial (i.e. approximate) answers to generally undecidable propositions and section 5 contains a worked example which illustrates the relationship between pre-conditions and conditioned slicing.

2 Theoretical Foundations

The thesis of this paper is that a program (which we call the 'Test Program') can be used as the answer to a 'question' about another program (which we call the 'Subject Program'). This 'question' also can be phrased as a program which tests a property of the state produced by the subject program. Of course, the property may only be required to hold when the subject program is executed in some set of well defined initial states. This means that we may in practice split the question into two parts, one which establishes the pre-condition on the initial state and one of which checks the post-condition on the final state.

The 'question program' is thus nothing more than a test harness. The crucial difference between the approach we advocate here and the traditional approach of testing a program using a test harness, is that conventional testing is *dynamic* whereas our approach is *static*. That is, rather than executing the Test Program with a series of inputs which satisfy the pre-condition and checking the corresponding post-condition, we simplify the Test Program, making it easier for a human to investigate the hypothesis manually.

We shall assume that the Test Harness tests propositions by storing either the value true or the value false in some identifiable variable. We shall write T_v for a Test Harness which stores its result in the variable v. A Test Harness can be composed with a subject program to form a Test Program. We shall write $\mathcal{C}_p^{T_v}$ to denote the composition of the Test Harness T_v with the subject program p. In practice this composition will either be sequential composition, or will simply consist of inserting the test program 'somewhere in the middle' of the Test Harness.

We refer to $\mathcal{M}$, the denotational meaning of programs. $\mathcal{M}[\![p]\!]$ is the state-to-state mapping denoted by the program p, where states are simply environment mappings from variable names to the values they denote. For a set of variables, V, we shall write $\mathcal{S}(p, V)$ for a simplified version of p which preserves its effect upon the variables in V. More formally, if p terminates when executed in some initial state σ then

$$\forall v.v \in V \Rightarrow \mathcal{M}[\![\mathcal{S}(p, V)]\!]\sigma v = \mathcal{M}[\![p]\!]\sigma v$$

This is a formalization of the semantic projection preserved by an end slice [11] constructed for the slicing criterion V.

In our approach, the programmer writes a Test Harness, T_v to test some property of a program p by storing a suitable value in a 'test variable', v. The Test Harness, T_v can then be composed with the Subject Program, p, to form the Test Program, $\mathcal{C}_p^{T_v}$. The system responds with $\mathcal{S}(\mathcal{C}_p^{T_v}, \{v\})$, a simplified version of the Test Program which preserves its effect upon the test variable.

Using this notation we may now define formally the relationship, which we call the 'Test Program Requirement (TPR)', which must exist between the Test Harness and the hypothesis it tests. It is the programmer's responsibility to meet the TPR by constructing a suitable Test Harness. If the TPR is met, the system will guarantee to produce a (hopefully) much simplified Test Program which also satisfies the TPR.

Definition 1 (Hypothesis) A hypothesis is a pair, containing two predicates which map states to booleans. The two predicates are the pre- and post-condition of the hypothesis

For example, let $\alpha = \lambda\sigma.\sigma(x) > 0$ and $\omega = \lambda\sigma.\sigma(z) < 0$, then the hypothesis (α, ω) states formally:

> "If the initial value of `x` is positive then the final value of `z` is negative."

Definition 2 (Test Program Requirement) Let $h = (\alpha, \omega)$ be a hypothesis.
Let T_v be a Test Harness that tests h, storing the result in the variable v.
The Test Program Requirement (TPR) is

$$\forall p.\forall\sigma.\mathcal{M}[\![p]\!]\sigma \neq \bot$$
$$\Rightarrow$$
$$\mathcal{M}[\![\mathcal{C}_p^{T_v}]\!]\sigma v = \begin{cases} \texttt{true} & \texttt{if } \alpha(\sigma) \Rightarrow \omega(\mathcal{M}[\![p]\!]\sigma) \\ \texttt{false} & \texttt{otherwise} \end{cases}$$

Observe that the TPR says nothing about non-terminating computations, and thus we are concerned only with the partial correctness of the hypothesis under test.

Now, $\mathcal{S}(p, V)$ is a simplified version of p which preserves its effect upon the variables in V. Since

$$\forall v.v \in V \Rightarrow \mathcal{M}[\![\mathcal{S}(p, V)]\!]\sigma v = \mathcal{M}[\![p]\!]\sigma v$$

We know that should $\mathcal{C}_p^{T_v}$ satisfy the TPR then $\mathcal{S}(\mathcal{C}_p^{T_v}, \{v\})$ will also do so.

Notice that $\alpha(\sigma) \Rightarrow \omega(\mathcal{M}[\![p]\!]\sigma)$ is not generally decidable, but neither is deciding whether $\mathcal{M}[\![\mathcal{C}_p^{T_v}]\!]\sigma v$ is true or false.

If the hypothesis is true for the program fragment under analysis, then the simplest program which will be obtainable, will be simply:

$$T \texttt{ = true;}$$

If the hypothesis is untrue for the program under investigation, then this will manifest itself as a simplified program (the simplified Test Program), which stores `false` in the Test Variable for at least one possible input to the Test Program.

To illustrate, suppose the program fragment is simply the single assignment

```
x = y*y;
```

If the hypothesis is that, in all states, this assignment stores $\texttt{y}^2$ in `x`, then the simplified test program will be

$$T \texttt{ = true;}$$

If, on the other hand, the hypothesis is that the program stores `y`$*$2 in `x`, then the simplest Test Program will be

$$\texttt{if (x==2) } T \texttt{ = true; else } T \texttt{ = false;}$$

Notice that this program suggests a pre-condition

$$\lambda\sigma.\sigma(x) = 2$$

Under this pre-condition, the post condition is satisfied.

As an example of how this approach might be used to investigate hypotheses during comprehension activity, consider the simple program fragment in Figure 2. In this figure the subject program is depicted on the left and the test program on the right. We have assumed for simplicity that the pre-condition is *true* and therefore there is no component of the test harness required to establish that the subject program is to be executed in a state which satisfies the precondition. (We shall return to this issue in section 5.)

The post-condition is that every element of the array `A` is no larger than the value stored in the variable `b`. More formally, the pre-condition is

$$\lambda\sigma.true$$

and the post-condition is[1]

$$\lambda\sigma.\forall i.0 \leq i < \texttt{MAX}.\sigma(\texttt{A}) \downarrow i \leq \texttt{b}$$

The hypothesis is that the program computes the biggest array value in `b`.

3 Test Program Simplification

The Test Program computes the answer that we are interested in, but it is larger than the program we started out with and so we have apparently gained little. However, we can simplify the Test Program, with the hope that the result *will* be (significantly) simpler than the original program.

[1] $\downarrow$ is the tuple selection operator.

1	`b=A[0];`	1	`b=A[0];`
2	`s=A[0];`	2	`s=A[0];`
3	`for(i=0;i<MAX;i++)`	3	`for(i=0;i<MAX;i++)`
4	`{ if(A[i]>b) b=A[i];`	4	`{ if(A[i]>b) b=A[i];`
5	`  if(A[i]<s) s=A[i]; }`	5	`  if(A[i]<s) s=A[i]; }`
		6	`T=true;`
		7	`for(i=0;i<MAX;i++)`
		8	`  if(A[i]>b) T=false;`
p: Original Program		t: Test program	

Figure 2: A Program and its Test Program

1	`b=A[0];`	1	`b=A[0];`
2	`s=A[0];`		
3	`for(i=0;i<MAX;i++)`	3	`for(i=0;i<MAX;i++)`
4	`{ if(A[i]>b) b=A[i];`	4	`  if(A[i]>b) b=A[i];`
5	`  if(A[i]<s) s=A[i]; }`		
6	`T=true;`	6	`T=true;`
7	`for(i=0;i<MAX;i++)`	7	`for(i=0;i<MAX;i++)`
8	`  if(A[i]>b) T=false;`	8	`  if(A[i]>b) T=false;`
t: Test Program		s: Slice on $(\{T\}, 9)$	

Figure 3: Slicing the Test Program

3.1 Slicing

An obvious way to tackle the size of the test program is to slice it with respect to the test variable. This is likely to produce simplification as the Test Harness will typically be designed to test only a *property* of the original program and therefore sections of the original will have no effect upon the variables mentioned in the Test Harness.

A program slice [14, 9] is constructed by deleting from a program those statements and predicates which cannot affect the value of a set of variables, V, at some point within the program, i. The pair (V, i) is known as the slicing criterion.

Several authors have shown how slicing may be used to aid the task of program comprehension and testing [2, 6, 10]. Slicing assists program comprehension because of the way it abstracts from a program a thread (the slice) which captures the effect of the original upon some chosen sub domain of its overall computation.

In this case we apply slicing to reduce the size of the Test Program, with the aim of reducing the effort required either to test it or to formally verify it. For this purpose we shall only be interested in end slicing [11], where the point of interest within the program to be sliced is the end of the program. An end slice on a set of variables, V, thus captures the portion of the program which affects the final values of the variables in V. In this case, V is simply the singleton set containing the test variable. The slice for the Test Program in Figure 2 is depicted in Figure 3.

3.2 Amorphous Slicing

Slicing the Test Program does reduce its size and 'weeds out' those parts of the original which are not concerned with the particular property under scrutiny. In the example we have just considered, it removes the computation of the smallest element in the array. However, traditional (syntax-preserving) slicing does not produce the smallest program which preserves the effect of the original upon the slicing criterion. In [7] an approach to slicing, called 'amorphous slicing', is introduced which retains the restriction that a slice preserves the effect of the original program with respect to the slicing criterion, but drops the requirement that a slice be constructed by command deletion alone.

An amorphous slice can therefore be constructed using any transformation which preserves the effect of the original upon the variables of interest. As such, an amorphous slice is no larger (and often considerably smaller) than the corresponding conventional slice. The price paid for this improved simplification power is that the slice produced is no longer syntactically related to the original (the process is 'syntactically amorphous', though semantically it is identical to conventional slicing).

For some applications, the syntactic relationship between a subject program and its conventional slice is crucial (for example for cohesion measurement [1, 13] and for debugging [12]). For other applications (such as the problem of program comprehension which presently concerns us) it is only the semantic relationship between slice and subject program which is important.

Using program transformation we can dramatically improve upon the slice we constructed for the Test Program in Figure 3. Figure 4 shows a possible sequence of transformations which yields the 'perfect' amorphous slice of the Test Program. We give this example as an illustration of the 'best case', to motivate our approach.

The transformations we use in Figure 4 are given in the form of a logical calculus in Figure 5. Three auxiliary functions are used in the definition of these rules. $\mathbf{DEF}(s)$ is the set of variables defined in statement s. Note that all our transformations are applicable only to programs in which expressions are side–effect free. That is, expressions have no defined variables. $\mathbf{REF}(s)$ is the set of variables referenced in statement or expression s. $\mathbf{SUB}(E, i, e)$ is the expression which results from substituting all occurrences of the identifier i, in the expression E, with expression e. $\mathbf{R_{EXPS}}(s)$ is the set of expressions mentioned by s (including those used to index arrays on either side of an assignment to an array element).

4 Programs as Approximate Answers

The Test Harness tests some aspect of the program. The Test Program which results from wrapping the Test Harness around the original program computes an answer to the hypothesis. Notice that no matter how complicated the syntax of the Test Program, it is semantically extremely simple; it assigns either `true` or `false` to the variable T. Of course, it is also likely that the syntax of the Test Program will be simpler, or, at least as simple as the original. The approach thus simplifies analysis of the hypothesis under investigation in two ways

- **Syntactically**
  Program slicing is used to reduce the shear amount of syntax to be considered.

- **Semantically**
  The test program's semantics is that of a program which stores a single boolean value in a variable.

5 Pre-conditions and Conditioned Slicing

The example in Figure 4 had a trivial pre-condition (*true*). In this section we consider amorphous slicing techniques which are applicable to Test Programs which embody a nontrivial pre-condition. Unsurprisingly, it turns out that what is required are techniques associated with conditioned slicing. Conditioned slicing was introduced by Canfora, Cimitile, DeLucia and Munro [3, 4]. The conditioned slicing criterion is an extension of the static slicing criterion introduced by Weiser [14].

A conditioned slicing criterion is a 4–tuple, (Υ, π, i, V), where π is a predicate logic formula, the free variables of which are Υ. V and i are the set of variables and the point in the program for which the slice is to be constructed (as in Weiser's static formulation).

A conditioned slice preserves the meaning of the original program on the set of variables in V when the next statement to be executed is at i and the program and its slice are executed in a state which satisfies π. The conditional slicing paradigm also extends naturally to amorphous conditioned slicing [7], where the restriction to command deletion is dropped in the slice construction process. By setting i to be the end of the program we obtain amorphous conditioned end slicing, which preserves the final values of variables in V when the program is executed in a state satisfying π.

To see how techniques associated with conditioned slicing have a role to play in slicing Test Programs with pre-conditions, consider the subject program and its Test Program in Figure 6.

In this program the pre-condition is

$$\lambda\sigma.\sigma(c) < 0$$

and the post-condition is

$$\lambda\sigma.\forall i.0 \leq i < MAX.\sigma(A) \downarrow i \geq 0$$

The pre-condition is established by enclosing the Test Program in the `then` branch of an `if` statement whose predicate is `c<0`, thereby leaving the subject program 'untested' if the pre-condition is not met (and leaving the value `true` in the result variable T).

Using the transformations from Figure 5 and conventional slicing we can transform the Test Program in Figure 6 to the simpler version depicted in Figure 7.

Axiom 1 $[\![e-e]\!] \;\Rightarrow\; [\![\mathtt{0}]\!]$

Axiom 2 $[\![e+\mathtt{0}]\!] \;\Rightarrow\; [\![e]\!]$

Axiom 3 $[\![i=i]\!] \;\Rightarrow\; [\![]\!]$

Axiom 4 $[\![e_1 \;\mathtt{\&\&}\; e_2]\!] \;\Rightarrow\; [\![e_2 \;\mathtt{\&\&}\; e_1]\!]$

Axiom 5 $[\![e \;\mathtt{\&\&}\; e]\!] \;\Rightarrow\; [\![e]\!]$

Axiom 6 $[\![E > E]\!] \;\Rightarrow\; [\![false]\!]$

Rule 1 (Push assign)

$$\frac{i_2 \notin \mathbf{REF}(e_1) \wedge e_3 = \mathbf{SUB}(e_2, i_1, e_1) \wedge i_1 \neq i_2}{[\![i_1 = e_1; i_2 = e_2;]\!] \Rightarrow [\![i_2 = e_3; i_1 = e_1;]\!]}$$

Rule 2 (Unfold assign)

$$\frac{e_3 = \mathbf{SUB}(e_2, i, e_1)}{[\![i = e_1; i = e_2;]\!] \Rightarrow [\![i = e_3;]\!]}$$

Axiom 7 (Then fold) $[\![\mathtt{if}\,(p)\; S_1 \quad S_2\;]\!] \Rightarrow [\![\mathtt{if}\,(p)\;\{S_1 S_2\}\;\mathtt{else}\; S_2]\!]$

Rule 3 (Redundant `if`)

$$\frac{p \equiv q}{[\![\mathtt{if}\,(p)\; S\;\mathtt{else}\;\;\mathtt{if}\,(q)\; S']\!] \Rightarrow [\![\mathtt{if}\,(p)\; S]\!]}$$

Rule 4 (Push `if`)

$$\frac{[\![i = e; S_2]\!] \Rightarrow [\![S_2'\; i = E;]\!] \wedge p' = \mathbf{SUB}(p, i, E)}{[\![i = E;\;\mathtt{if}\,(p)\; S_2]\!] \Rightarrow [\![\mathtt{if}\,(p')\; S_2'\; i = E;\;]\!]}$$

Rule 5 (Loop coalesce)

$$\frac{\forall e \in \mathbf{R_{EXPS}}(S_1).\forall e' \in \mathbf{R_{EXPS}}(S_2).i \in \mathbf{REF}(e') \wedge i \in \mathbf{REF}(e) \Rightarrow [\![e\mathtt{<=}e']\!] \equiv [\![\mathtt{true}]\!]}{[\![\mathtt{for}\,(i = e_1; e_2; i{+}{+})\; S_1\;\;\mathtt{for}\,(i = e_1; e_2; i{+}{+})\; S_2]\!] \Rightarrow [\![\mathtt{for}\,(i = e_1; e_2; i{+}{+})\;\{S_1 S_2\}\;]\!]}$$

Rule 6 (Code motion)

$$\frac{\mathbf{REF}(S_1) \cap \mathbf{DEF}(S_2) = \emptyset \;\;\wedge\;\; \mathbf{REF}(S_2) \cap \mathbf{DEF}(S_1) = \emptyset}{[\![S_1 S_2]\!] \Rightarrow [\![S_2 S_1]\!]}$$

Axiom 8 (`if` reduce false) $[\![\mathtt{if}\,(\mathtt{false})\; S_1\;\mathtt{else}\; S_2]\!] \Rightarrow [\![S_2]\!]$

Figure 5: Transformation Axioms and Rules

p: Original Program

```
   for(i=0;i<MAX;i++)
     if(A[i]<0)
       A[i]=A[i]*c;
```

t: Test program

```
T=true;
if(c<0) {

   for(i=0;i<MAX;i++)
     if(A[i]<0)
       A[i]=A[i]*c;

   for(i=0;i<MAX;i++)
     if(A[i]<0)
       T=false; }
```

Figure 6: A Test Program with a Pre-condition

Now, using the path condition approach employed in the construction of conditioned slices [3], we can establish that the path condition at the statement

`if((A[i]*c)<0) T=false;`

is

$$\texttt{c<0} \;\wedge\; \texttt{A[i]<0}$$

In the conditioned slicing paradigm, such path conditions are used to simplify conditional statements, where the truth or falsity of the predicate guard can be inferred from the path condition. In this case, using simple theorem proving, we can establish

$$\texttt{c<0} \wedge \texttt{A[i]<0} \Rightarrow \texttt{!(A[i]*c} < \texttt{0)}$$

thereby allowing us to remove the statement

`if((A[i]*c)<0) T=false;`

from the Test Program. The only remaining statement which affects the final value of T is now the initial assignment T`=true;`, and so we have reached a final answer (a minimal slice).

Of course we have, once again, chosen an illustrative example where it is relatively easy to produce a minimal slice. In general, we may not be so fortunate.

6 Conclusion

We have introduced a philosophy of 'programs as answers to undecidable propositions', in which hypotheses concerning a subject program are themselves, expressed as programs. The truth or otherwise of these hypotheses is also represented as a program, allowing us to go some way towards the formation of a bridge between formal program verification and program testing, much in the spirit of Gaudel [5].

In particular, we argue that the answer (whether the hypothesis holds or not) can be approximated using program simplification technologies such as conditioned and amorphous slicing.

The approximate nature of the approach addresses the undecidable nature of general program hypotheses, providing a delineation between what can be expected of an automated formal manipulation system and what remains in the realm of human (non-automated or semi-automated) investigation.

Acknowledgements

David Binkley is supported by National Science Foundation grant CCR-9803665. The authors would also like to acknowledge the insight provided by several conversations with Gerardo Canfora, Andrea DeLucia, Jeremy Dick, Keith Gallagher, Arun Lakhotia, Malcolm Munro and Hongji Yang which helped formulate much of the work reported here.

References

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

[2] Binkley, D. W. The application of program slicing to regression testing. In *Journal of Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier Science B. V., 1998. to appear.

[3] Canfora, G., Cimitile, A., and De Lucia, A. Conditioned program slicing. In *Journal of Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier Science B. V., 1998. to appear.

[4] Cimitile, A., De Lucia, A., and Munro, M. A specification driven slicing process for identifying

⇒ (Code motion)

```
b=A[0]; T=true;
for(i=0;i<MAX;i++)
  if(A[i]>b) b=A[i];
for(i=0;i<MAX;i++)
  if(A[i]>b) T=false;
```

⇒ (Loop Coalesce)

```
b=A[0]; T=true;
for(i=0;i<MAX;i++)
{ if(A[i]>b) b=A[i];
  if(A[i]>b) T=false;}
```

⇒ (Then fold)

```
b=A[0]; T=true;
for(i=0;i<MAX;i++)
  if(A[i]>b)
  { b=A[i];
    if(A[i]>b) T=false;}
  else if(A[i]>b) T=false;
```

⇒ (redundant if)

```
b=A[0]; T=true;
for(i=0;i<MAX;i++)
  if(A[i]>b)
  { b=A[i];
    if(A[i]>b) T=false;}
```

⇒ (push if)

```
b=A[0]; T=true;
for(i=0;i<MAX;i++)
  if(A[i]>b)
  { if(A[i]>A[i]) T=false;
    b=A[i];}
```

⇒ (axioms & if reduce false)

```
b=A[0]; T=true;
for(i=0;i<MAX;i++)
  if(A[i]>b) b=A[i];
```

⇒ (Slice on T)

```
T=true;
```

Figure 4: Test Program Transformation

```
T=true;
if(c<0) {
  for(i=0;i<MAX;i++)
    if(A[i]<0) {
      if((A[i]*c)<0)
        T=false;
      A[i]=A[i]*c; }}
```

Figure 7: Simplified Test Program

reusable functions. *Software maintenance: Research and Practice 8* (1996), 145–178.

[5] Gaudel, M.-C. Testing can be formal, too. *Lecture Notes in Computer Science 915* (1995), 82–96.

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

[7] Harman, M., and Danicic, S. Amorphous program slicing. In 5^{th} *IEEE Internation Workshop on Program Comprehesion (IWPC'97)* (Dearborn, Michigan, USA, May 1997), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 70–79.

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

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

[10] Jackson, D., and Rollins, E. J. Chopping: A generalisation of slicing. Tech. Rep. CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1994.

[11] Lakhotia, A. Rule–based approach to computing module cohesion. In *Proceedings of the* 15^{th} *Conference on Software Engineering (ICSE-15)* (1993), pp. 34–44.

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

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

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