ILLUSTRATING INTERFERENCE IN INTERFERING VERSIONS OF PROGRAMS

by

Thomas Reps and Thomas Bricker

Computer Sciences Technical Report #827

March 1989

Illustrating Interference in Interfering Versions of Programs

THOMAS REPS and THOMAS BRICKER
University of Wisconsin – Madison

The need to integrate several versions of a program into a common one arises frequently, but it is a tedious and time consuming task to merge programs by hand. The program-integration algorithm recently proposed by S. Horwitz, J. Prins, and T. Reps provides a way to create a *semantics-based* tool for program integration. The integration algorithm is based on the assumption that any change in the *behavior*, rather than the *text*, of a program variant is significant and must be preserved in the merged program. An integration system based on this algorithm will either automatically combine several different but related variants of a base program, or else determine that the programs incorporate interfering changes.

In this paper we discuss how an integration tool can illustrate the causes of interference to the user when interference is detected. Our main technical result is an alternative characterization of the integration algorithm's interference criterion that is more suitable for illustrating the causes of interference. We then propose six methods for an integration system to display information to demonstrate the causes of interference to the user.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques – *programmer workbench*; D.2.3 [**Software Engineering**]: Coding – *program editors*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution and Maintenance – *enhancement, restructuring, version control*; D.2.9 [**Software Engineering**]: Management – *programming teams, software configuration management*; D.3.4 [**Programming Languages**]: Processors; E.1 [**Data Structures**] *graphs*

General Terms: Theory

Additional Key Words and Phrases: program integration, interference, dependence graph, program slicing, control dependence, data dependence

1. INTRODUCTION

The program-integration algorithm recently proposed by S. Horwitz, J. Prins, and T. Reps provides a way to create a *semantics-based* tool for program integration [1-3]. Semantics-based integration is based on the assumption that a difference in the *behavior* of one of the variant programs from that of the base program, rather than a difference in the *text*, is significant and must be preserved in the merged program. Although it is undecidable to determine whether a program modification actually leads to such a difference, it is possible to determine a safe approximation by comparing each of the variants with the base program.

To determine this information, the integration algorithm employs a program representation that is similar (although not identical) to the *dependence graphs* that have been used previously in vectorizing and paral-

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grant DCR-8552602, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from IBM, DEC, and Xerox.

Authors' address: Computer Sciences Department, University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706.

lelizing compilers. The algorithm also makes use of the notion of a *program slice* to find just those statements of a program that determine the values of potentially affected variables, as well as to characterize interfering variants.[1]

In this paper, we describe an alternative characterization of the algorithm's interference criterion; the new interference test presented here is more suitable for illustrating the causes of interference when interference is detected. Our description of the new test assumes familiarity with the technical details of the program-integration algorithm; the reader is referred to [1-3] for definitions of the terms used in this paper and for a description of the steps of the program-integration algorithm.

A preliminary implementation of a program-integration tool that uses the technique from [1-3] has been embedded in a program editor created using the Synthesizer Generator, a meta-system for creating interactive, language-based program development systems [4,5]. Data-flow analysis of programs is carried out according to the editor's defining attribute grammar and used to construct dependence graphs. An integration command added to the editor invokes the integration algorithm on the dependence graphs, reports whether the variant programs interfere, and, if there is no interference, builds the integrated program. Several of the methods for illustrating interference described below have been incorporated into this tool.

The remainder of the paper is divided into three sections. Section 2 introduces some terminology that is used in the new characterization of interference. Section 3 defines the *interference vertices* arising from the integration of two program variants with respect to a base program and shows that they may be used to characterize interference. Section 4 describes six methods for an integration system to display information to demonstrate the causes of interference to the user.

2. AFFECTED POINTS AND DIRECTLY AFFECTED POINTS

We define the sets of edges incident on a vertex v in dependence graph G as follows:

$\text{IncidentControl}(v, G) = \{ w \rightarrow_c v \mid w \rightarrow_c v \in E(G) \}$

$\text{IncidentFlow}(v, G) = \{ w \rightarrow_f v \mid w \rightarrow_f v \in E(G) \}$

$\text{IncidentDefOrder}(v, G) = \{ x \rightarrow_{do(v)} y \mid x \rightarrow_{do(v)} y \in E(G) \}$

The set $\text{IncidentFlow}(v, G)$ can be further subdivided into $\text{IncidentLoopIndependentFlow}(v, G)$ and $\text{IncidentLoopCarriedFlow}(v, G)$. Note that a def-order edge $x \rightarrow_{do(v)} y$ can be thought of as a hyper-edge directed from x to y to v. It is in this sense that a def-order edge is incident on witness vertex v.

Definition. Given dependence graphs G_M and G_N, the set $DAP_{M,N}$ of vertices in G_M that are *directly affected* with respect to G_N, consisting of all vertices of G_M that have different incident-edge sets than the corresponding vertices of G_N, is

$$DAP_{M,N} = \{ v \in V(G_M) \mid \text{IncidentControl}(v, G_M) \neq \text{IncidentControl}(v, G_N) \vee \text{IncidentFlow}(v, G_M) \neq \text{IncidentFlow}(v, G_N) \vee \text{IncidentDefOrder}(v, G_M) \neq \text{IncidentDefOrder}(v, G_N) \}.$$

We can give a definition of $AP_{M,N}$, the affected points of G_M with respect to G_N, in terms of the directly affected points of G_M with respect to G_N using the following concept:

---

[1]It should be noted, however, that the integration capabilities of the tool are severely limited; in particular, the tool can only handle programs written in a simple language in which expressions contain scalar variables and constants, and the only statements are assignment statements, conditional statements, and while-loops.

Definition. Given dependence graph G and vertex set S, the set AffectedBy(S, G), consisting of vertices *affected by* members of S is

AffectedBy$(S, G) = \{ v \in V(G) \mid S \cap V(G / v) \neq \emptyset \}$.

The set AffectedBy(S, G) is the vertex set of the "forward slice" of G with respect to S. Thus, $AP_{M,N} = \text{AffectedBy}(DAP_{M,N}, G_M)$.

3. INTERFERENCE VERTICES: AN ALTERNATIVE CHARACTERIZATION OF INTERFERENCE

It is sometimes convenient to use one of a number of alternative characterizations of the interference condition

$(G_A / AP_{A,Base} \neq G_M / AP_{A,Base})$ or $(G_B / AP_{B,Base} \neq G_M / AP_{B,Base})$

(so called "Type I" interference[2]). It is possible to characterize the Type I interference condition in terms of a certain set of vertices (say S) and a statement of the form: "There is Type I interference if and only if the set S is non-empty."

One characterization of this form is that there is Type I interference if and only if the following set is non-empty:

$(AP_{A,Base} \cap AP_{M,A}) \cup (AP_{B,Base} \cap AP_{M,B})$.

This is shown by observing that the clause "$G_A / AP_{A,Base} \neq G_M / AP_{A,Base}$" is equivalent to "The set $AP_{A,Base} \cap AP_{M,A}$ is non-empty." Both conditions say that there is some vertex $v \in AP_{A,Base}$ such that $G_A / v \neq G_M / v$.

Our method for illustrating interference when integration fails is based on yet another characterization of the Type I interference condition. The new characterization is based on the notion of *interference vertices*, defined as follows:

Definition. The set of *interference vertices* arising from the integration of programs A and B with respect to *Base* is the set

InterferenceVertices$(A, B, Base) = (V(G_A / AP_{A,Base}) \cap DAP_{M,A}) \cup (V(G_B / AP_{B,Base}) \cap DAP_{M,B})$.

The theorem proven below shows that the integration of A and B with respect to *Base* leads to Type I interference if and only if the set of interference vertices InterferenceVertices$(A, B, Base)$ is non-empty.

THEOREM (Alternative Characterization of Type I Interference). *The integration of A and B with respect to Base leads to Type I interference if and only if the set* InterferenceVertices$(A, B, Base)$ *is non-empty.*

PROOF.

$\Rightarrow$ case:

Without loss of generality, assume that G_A exhibits the interference (*i.e.* $G_A / AP_{A,Base} \neq G_M / AP_{A,Base}$). This is equivalent to saying that there is some vertex $w \in AP_{A,Base}$ such that $G_M / w \neq G_A / w$. Working back from w in graph G_A / w we must eventually come to a vertex v for which the set of incident control,

---

[2]The Type II interference condition is that the merged dependence graph G_M is infeasible; that is, there is no program P whose dependence graph G_P is identical to G_M.

flow, or def-order edges are different in G_M than in G_A. Because $v \in V(G_A / AP_{A,Base})$ and $G_A / AP_{A,Base}$ is a subgraph of G_M, $v \in V(G_M)$. Thus, by definition, vertex v is a member of $DAP_{M,A}$. We have now shown that if G_A exhibits the interference there is a vertex v such that $v \in V(G_A / AP_{A,Base})$ and $v \in DAP_{M,A}$ (*i.e.*, the set $V(G_A / AP_{A,Base}) \cap DAP_{M,A}$ is non-empty).

Because the same argument applies if G_B exhibits the interference, we conclude that, if there is interference, the set $(V(G_A / AP_{A,Base}) \cap DAP_{M,A}) \cup (V(G_B / AP_{B,Base}) \cap DAP_{M,B})$ is non-empty.

$\Leftarrow$ case:

Assume that InterferenceVertices$(A, B, Base) \neq \emptyset$, and let v be a member of InterferenceVertices$(A, B, Base)$. Without loss of generality, assume that $v \in (V(G_A / AP_{A,Base}) \cap DAP_{M,A})$. But $G_M / v \neq G_A / v$, because any member of $DAP_{M,A}$ must have a different slice in G_M than in G_A, which means that the slice $V(G_A / AP_{A,Base})$ is not preserved in G_M. We conclude that A and B interfere with respect to $Base$. □

An important property of the characterization of Type I interference in terms of interference vertices is that InterferenceVertices$(A, B, Base) \subseteq (DAP_{A,Base} \cup DAP_{B,Base})$. That is, the members of InterferenceVertices$(A, B, Base)$ are members of either $DAP_{A,Base}$ or $DAP_{B,Base}$ (or both), as shown below in the proof of the following theorem.

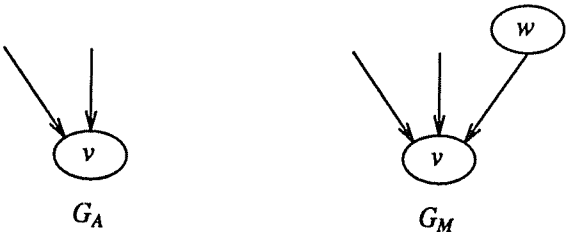
THEOREM (DAP-Union Theorem). InterferenceVertices$(A, B, Base) \subseteq (DAP_{A,Base} \cup DAP_{B,Base})$.

PROOF. Without loss of generality, assume that G_A exhibits the interference, which is to say that the set $V(G_A / AP_{A,Base}) \cap DAP_{M,A}$ is non-empty. That is, there is a vertex v such that $v \in V(G_A / AP_{A,Base})$ and $v \in DAP_{M,A}$.

What remains to be shown is that either $v \in DAP_{A,Base}$ or $v \in DAP_{B,Base}$. Supposing that $v \notin DAP_{A,Base}$, there are four cases to consider:

Case 1.

Suppose that in G_M v is the target of a flow edge (respectively, control edge) from vertex w, where edge $w \rightarrow_f v$ $(w \rightarrow_c v)$ is not found in G_A.

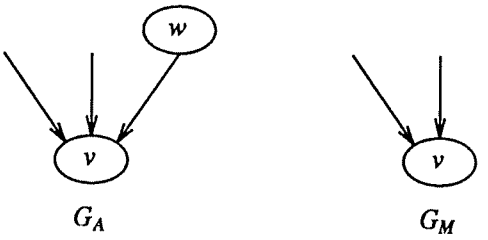


G_A G_M

Since $v \notin DAP_{A,Base}$, G_{Base} must contain the same flow (control) edges into v as are found in G_A. In particular, the edge $w \rightarrow_f v$ $(w \rightarrow_c v)$ is not in the edge set of G_{Base}. From the construction of G_M, it must be that edge $w \rightarrow_f v$ $(w \rightarrow_c v)$ is a flow (control) edge of G_B, which implies that $v \in DAP_{B,Base}$.

Case 2.

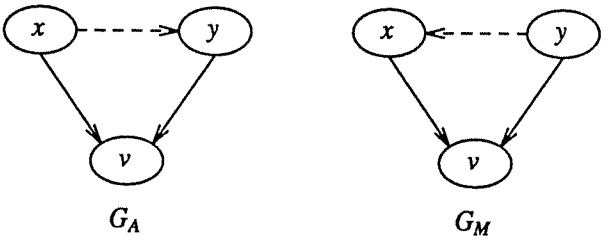
Suppose that in G_A v is the target of a flow edge (respectively, control edge) from vertex w, where edge $w \rightarrow_f v$ $(w \rightarrow_c v)$ is not found in G_M.



But by the construction of G_M, this is not possible because $v \in V(G_A)/AP_{A,Base}$ and $V(G_A)/AP_{A,Base}$ is a subgraph of G_M. (By the definition of the edge set of a slice, in G_M v must have all of the incoming control or flow predecessors that v has in G_A.)

Case 3.

Suppose that in G_M v is the witness of a def-order edge $y \longrightarrow_{do(v)} x$ not found in G_A.



By Cases (1) and (2), for v not to be an element of $DAP_{A,Base}$, G_A must contain edges $x \longrightarrow_f v$ and $y \longrightarrow_f v$. Given that these edges are in G_A and that edge $y \longrightarrow_{do(v)} x$ is not in G_A, it must be that a def-order edge $x \longrightarrow_{do(v)} y$ occurs in G_A and hence in G_{Base}. By the construction of G_M, the only way edge $y \longrightarrow_{do(v)} x$ could occur in G_M is if $y \longrightarrow_{do(v)} x$ occurs in G_B. Because G_{Base} cannot contain both $x \longrightarrow_{do(v)} y$ and $y \longrightarrow_{do(v)} x$, this implies that $v \in DAP_{B,Base}$.

Case 4.

Suppose that in G_A v is the witness of a def-order edge $x \longrightarrow_{do(v)} y$ not found in G_M. But by the construction of G_M, this is not possible because $v \in V(G_A)/AP_{A,Base}$ and $V(G_A)/AP_{A,Base}$ is a subgraph of G_M. (By the definition of the edge set of a slice, in G_M v must be a witness for all def-order edges for which v is a witness in G_A.)

□

A second important property of the characterization of Type I interference in terms of interference vertices is that InterferenceVertices(A, B, $Base$) $\subseteq (V(G_A) \cap V(G_B))$; that is, the members of InterferenceVertices(A, B, $Base$) are members of both $V(G_A)$ and $V(G_B)$, as shown below in the proof of the following theorem.

THEOREM (AB-Intersection Theorem). InterferenceVertices(A, B, $Base$) $\subseteq (V(G_A) \cap V(G_B))$.

PROOF. Suppose v is a member of

$$(V(G_A/AP_{A,Base}) \cap DAP_{M,A}) \cup (V(G_B/AP_{B,Base}) \cap DAP_{M,B}).$$

Without loss of generality, assume that $v \in (V(G_A/AP_{A,Base}) \cap DAP_{M,A})$. From inspection of the first subterm, it is obvious that $v \in V(G_A)$. Because $v \in DAP_{M,A}$ we know that there is an edge incident on v in G_M that does not occur in G_A. By the construction of G_M, this edge can only come from the edge set of G_B; consequently, $v \in V(G_B)$. □

4. DISPLAYING INTERFERING SLICES

Interference vertices provide a useful basis for illustrating Type I interference because of the two properties captured by the DAP-Union Theorem and the AB-Intersection Theorem:

(1) By the AB-Intersection Theorem, interference vertices are members of both $V(G_A)$ and $V(G_B)$. Consequently, interference can be illustrated by displaying information about interference vertices *simultaneously* in programs A and B.

(2) By the DAP-Union Theorem, each interference vertex is a directly-affected element in at least one of the programs A and B. For this reason, an interference vertex identifies a *direct effect* of a change introduced by at least *one* of the programmers.

Interference vertices can be used in several ways to illustrate Type I interference; six methods are described below, all of which are based on making a comparison of slices of G_A and G_B with respect to interference vertices.

In devising methods for illustrating interference, a practical consideration is whether the amount of information presented will overwhelm the user. Thus, of the methods discussed below, we feel that Methods 5 and 6, which break down the information to the finest level of granularity, are the ones that are most likely to be useful to users who are trying to diagnose the cause of interference in a failed integration.

We will demonstrate our methods for illustrating interference with a running example based on the three program versions shown below. Buffer demoBase contains a program to sum the integers from 1 to 10. Buffer demoA contains a version of the program in buffer demoBase (created by editing a *copy* of demoBase). It differs from the program in demoBase in two respects:

(1) There is an additional statement at the end of the program, amean := sum / i, which computes the arithmetic mean.

(2) The initialization statement i := 1 has been changed to i := 0.

Buffer demoB contains a second version of the program in buffer demoBase, incorporating the computation of the geometric mean (but not the arithmetic mean, which was introduced solely in demoA).

demoA

```
program demo;
begin
   sum := 0;
   i := 0;
   while (i <= 10) do
      sum := sum + i;
      i := i + 1
   od;
   amean := sum / i
end.
```

demoBase

```
program demo;
begin
   sum := 0;
   i := 1;
   while (i <= 10) do
      sum := sum + i;
      i := i + 1
   od;
end.
```

demoB

```
program demo;
begin
   sum := 0;
   prod := 1;
   i := 1;
   while (i <= 10) do
      sum := sum + i;
      prod := prod * i;
      i := i + 1
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

In this example, if we try to integrate the programs in buffers demoA and demoB with respect to the base program in buffer demoBase, the interference vertices are the loop predicate (i <= 10) and the statement i := i + 1.

(1) Predicate (i <= 10) is an interference vertex because it is directly affected with respect to demoBase in the dependence graph of demoA and directly affected with respect to demoA in the merged dependence graph. Note that in the merged dependence graph, (i <= 10) has incident loop-independent flow edges from both statement i := 0 and statement i := 1.

(2) Statement i := i + 1 is an interference vertex for essentially the same reasons as predicate (i <= 10); i := i + 1 is directly affected with respect to demoBase in the dependence graph of demoA and directly affected with respect to demoA in the merged dependence graph. Note that in the merged dependence graph, i := i + 1 has incident loop-independent flow edges from both statement i := 0 and statement i := 1.

Method 1:

When integration fails due to Type I interference, we can illustrate interference by displaying the slices G_A / InterferenceVertices(A, B, $Base$) and G_B / InterferenceVertices(A, B, $Base$). These slices represent computation threads that need to be preserved in the merged dependence graph G_M, but which the integration algorithm does not (and cannot) preserve. Note that these slices are taken with respect to vertices that are directly-affected points (with respect to G_{Base}) of G_A, G_B, or both.

Example. The slices of demoA and demoB with respect to the two interference vertices of our running example (loop predicate (i <= 10) and statement i := i + 1) are shown below. In the screen images shown below, elements of a slice are indicated by enclosing them in double angle brackets — << and >>; obviously other mechanisms, including color, could be employed to make the elements of a slice stand out better from the rest of the program.

demoA

```
program demo;
begin
   sum := 0;
   <<i := 0>>;
   while <<(i <= 10)>> do
      sum := sum + i;
      <<i := i + 1>>
   od;
   amean := sum / i
end.
```

demoB

```
program demo;
begin
   sum := 0;
   prod := 1;
   <<i := 1>>;
   while <<(i <= 10)>> do
      sum := sum + i;
      prod := prod * i;
      <<i := i + 1>>
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

Method 2:

Instead of displaying in their entirety both of the slices G_A / InterferenceVertices(A, B, $Base$) and G_B / InterferenceVertices(A, B, $Base$), it is possible to display slices of G_A and G_B with respect to single interference vertices (*i.e.* individual vertices v, where $v \in$ InterferenceVertices(A, B, $Base$)). Commands can be provided in the integration system to permit the user to step through all the possible v's and display the slices G_A / v and G_B / v.

Example. To illustrate the interference that arises when demoA and demoB are integrated with respect to demoBase, we invoke a command to move the respective selections of buffers demoA and demoB to an interference vertex and display the slice of the buffer with respect to that vertex. For instance, because loop predicate (i <= 10) is one of the interference vertices of the example, when this command is applied the two selections would be moved to (i <= 10), and their slices with respect to (i <= 10) would be displayed, as shown below. (A buffer's selection is indicated by outlining it in a single-ruled box.)

demoA

```
program demo;
begin
   sum := 0;
   <<i := 0>>;
   while <<(i <= 10)>> do
      sum := sum + i;
      <<i := i + 1>>
   od;
   amean := sum / i
end.
```

demoB

```
program demo;
begin
   sum := 0;
   prod := 1;
   <<i := 1>>;
   while <<(i <= 10)>> do
      sum := sum + i;
      prod := prod * i;
      <<i := i + 1>>
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

The significance of a slice is that it captures a portion of a program's behavior in the sense that, for any initial state on which the program halts, the program and the slice compute the same sequence of values for each element of the slice [6]. In our case a program point may be an assignment statement or a control predicate. Because a statement or control predicate may be reached repeatedly in a program, by "computing the same sequence of values for each element of the slice" we mean: (1) for an assignment statement the same *sequence* of values is assigned to the target variable; (2) for a predicate the same *sequence* of boolean values is produced.

Because the slices shown above are not the same, predicate (i <= 10) may take on a different sequence of values in an execution of demoA than it does in an execution of demoB; (i <= 10) is an interference vertex because the two slices cannot both be preserved in the merged dependence graph.

Continuing the example, when the command to show the next interference vertex is given, the respective selections of demoA and demoB are changed to statement i := i + 1, and the slices shown below would be displayed in the two buffers. As before, these represent slices that cannot both be preserved in the merged dependence graph.

```
demoA
program demo;
begin
   sum := 0;
   <<i := 0>>;
   while <<(i <= 10)>> do
      sum := sum + i;
      <<i := i + 1>>
   od;
   amean := sum / i
end.
```

```
demoB
program demo;
begin
   sum := 0;
   prod := 1;
   <<i := 1>>;
   while <<(i <= 10)>> do
      sum := sum + i;
      prod := prod * i;
      <<i := i + 1>>
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

Method 3:

For each interference vertex v, it is also possible to provide information about what is different between the slices G_A / v and G_B / v. Rather than displaying the slices themselves, it is possible to point to the particular edges that make the two slices different. Thus, if v is an interference vertex, the system can display in programs A and B the endpoints of all edges in the symmetric difference of the slices' edge sets (*i.e.*, $E(G_A / v) \Delta E(G_B / v)$). More precisely, the edge-set difference $E(G_A / v) - E(G_B / v)$ is displayed in program A and the edge-set difference $E(G_B / v) - E(G_A / v)$ is displayed in program B.

Example. In the slice of buffer demoA with respect to interference vertex (i <= 10) there is a flow edge from i := 0 to (i <= 10) that does not occur in the slice of demoB with respect to (i <= 10). Likewise, in the slice of buffer demoB with respect to (i <= 10) there is a flow edge from i := 1 to (i <= 10) that does not occur in the slice of demoA with respect to (i <= 10). As shown below, this difference can be illustrated by displaying the endpoints of the offending edges. Endpoints of individual dependence edges are indicated below by enclosing them in double square brackets — [[and]].

```
demoA
program demo;
begin
   sum := 0;
   [[i := 0]];
   while [[(i <= 10)]] do
      sum := sum + i;
      i := i + 1
   od;
   amean := sum / i
end.
```

```
demoB
program demo;
begin
   sum := 0;
   prod := 1;
   [[i := 1]];
   while [[(i <= 10)]] do
      sum := sum + i;
      prod := prod * i;
      i := i + 1
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

Continuing the example, in the slice of buffer demoA with respect to interference vertex i := i + 1 there is a flow edge from i := 0 to i := i + 1 that does not occur in the slice of demoB with respect to i := i + 1. Likewise, in the slice of buffer demoB with respect to i := i + 1 there is a flow edge from i := 1 to i := i + 1 that does not occur in the slice of demoA with respect to i := i + 1. The endpoints of the offending edges are shown below.

demoA

```
program demo;
begin
   sum := 0;
   [[i := 0]];
   while (i <= 10) do
      sum := sum + i;
      [[i := i + 1]]
   od;
   amean := sum / i
end.
```

demoB

```
program demo;
begin
   sum := 0;
   prod := 1;
   [[i := 1]];
   while (i <= 10) do
      sum := sum + i;
      prod := prod * i;
      [[i := i + 1]]
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

It is instructive to consider why we choose to display the symmetric difference of the *edge sets* rather than the symmetric difference of the *vertex sets*. To see why, consider the following example:

demo2A

```
program demo2;
begin
   x := 0;
   if P then
      x := 1
   fi;
   if Q then
      x := 2
   fi;
   y := x
end.
```

demo2Base

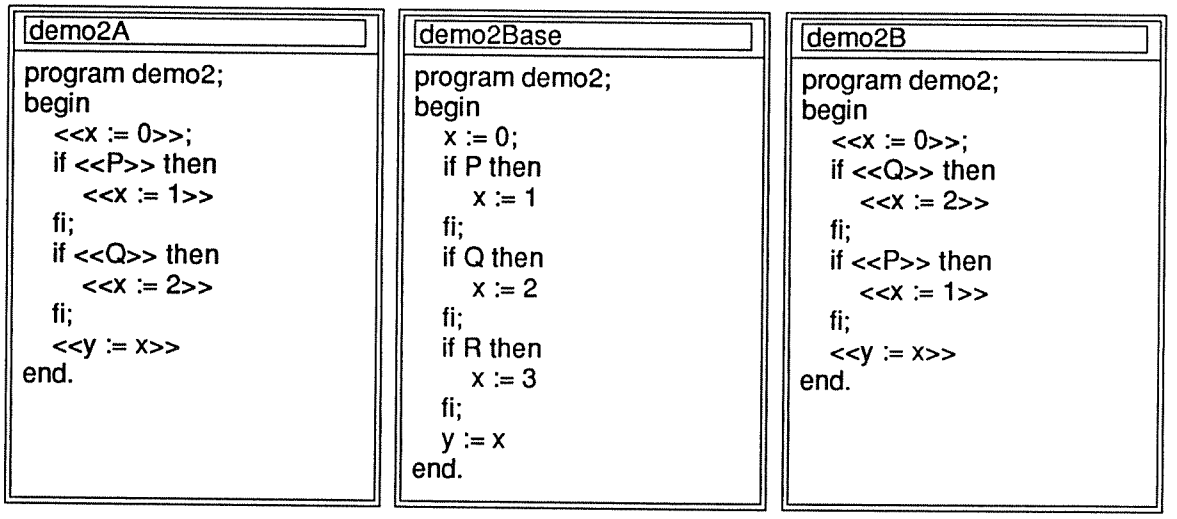
```
program demo2;
begin
   x := 0;
   if P then
      x := 1
   fi;
   if Q then
      x := 2
   fi;
   if R then
      x := 3
   fi;
   y := x
end.
```

demo2B

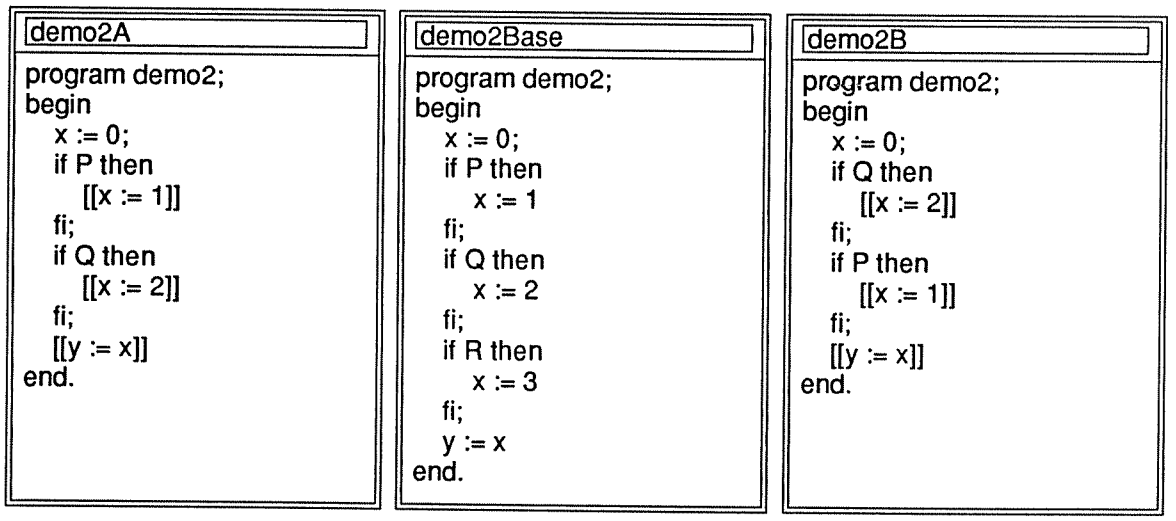
```
program demo2;
begin
   x := 0;
   if Q then
      x := 2
   fi;
   if P then
      x := 1
   fi;
   y := x
end.
```

In this example, the statement if R then x := 3 fi has been deleted from both the programs in buffers demo2A and demo2B. The only other change is in buffer demo2B, where the order of statements if P then x := 1 fi and if Q then x := 2 fi has been interchanged; this change is a *rearrangement* of program elements that also exist in buffers demo2A and demo2Base and introduces no *new* statements.

The vertex that corresponds to the assignment statement y := x is the sole interference vertex in this example. However, the slices G_A / v and G_B / v, shown below, have the same vertex sets; consequently, the symmetric difference of the vertex sets, $V(G_A / v) \Delta V(G_B / v)$, is empty.



By contrast, the symmetric difference of the *edge sets* is non-empty: the difference $E(G_A / v) - E(G_B / v)$ consists of the def-order edge [x := 1] $\rightarrow_{do}$ [x := 2] (with witness y := x); the difference $E(G_B / v) - E(G_A / v)$ consists of the edge [x := 2] $\rightarrow_{do}$ [x := 1] (also with witness y:= x). Thus, the interference in this example would be illustrated by displaying the endpoints of these two (hyper-)edges, as follows:



Note that in the case of Type I interference due to a flow edge in G_A (or G_B) from an entirely new vertex, the new vertex is at the tail of a new edge, so the edge-set-difference method and a method for illustrating interference based on vertex-set differences would display almost the same information. However, the edge-set-difference method also handles cases like the one discussed above in which the vertex-set difference is empty.

Method 4:

Because Method 3 displays all edges of the edge-set difference $E(G_A / v) \Delta E(G_B / v)$ simultaneously, Method 3 may overwhelm the user with too much information. Furthermore, the information displayed with Method 3 does not make it clear how edges displayed in program A relate to edges displayed in program B, and *vice versa*. Thus, instead of displaying all edges of the set $E(G_A / v) \Delta E(G_B / v)$ simultane-

ously, Method 4 displays only the edges of the set that are incident on a single vertex w; commands are provided to step through all such w's.

Example. For each of the two interference vertices in the example that accompanies Method 3 (predicate (i <= 10) and statement i := i + 1), there is only one edge in each of the set differences $E(G_A / v) - E(G_B / v)$ and $E(G_B / v) - E(G_A / v)$. For interference vertex (i <= 10) both of the edges are incident on (i <= 10); for interference vertex i := i + 1 both of the edges are incident on i := i + 1. Thus, for this example the information that would be displayed using Method 4 is exactly the same as that displayed using Method 3.

Method 5:

A further refinement on Method 4 is to break down the information to a finer level of granularity and display separately the different *classes* of edges that are incident on w. That is, instead of displaying simultaneously all four kinds of edges incident on w that are members of $E(G_A / v) \Delta E(G_B / v)$, different commands are provided to display separately the control edges, loop-independent flow edges, loop-carried flow edges, and def-order edges.

Example. For each of the two interference vertices in the example that accompanies Method 3, the edges in the set differences $E(G_A / v) - E(G_B / v)$ and $E(G_B / v) - E(G_A / v)$ are loop-independent flow edges. Thus, for this example, when loop-independent flow edges are displayed, the information displayed using Method 5 is exactly the same as that displayed using Method 3.

Method 6:

For each vertex $v \in \text{InterferenceVertices}(A, B, Base))$, it is possible to provide information about why a particular edge that Method 5 would bring to light (in one program) conflicts with the changed computation threads of the other program. Thus, Method 6 displays in program A an edge (y, z) in the edge difference $E(G_A / v) - E(G_B / v)$ while at the same time displaying in program B a slice G_B / w, where w is a vertex of B that is not only affected by v but is also one of the affected points of B with respect to $Base$. That is, w is a vertex such that $w \in (\text{AffectedBy}(\{ v \}, G_B) \cap AP_{B,Base})$.

Example. In our running example, interference vertex (i <= 10) affects statement gmean := prod ** (1 / (i – 1)), which is also an affected point of demoB with respect to demoBase. Thus, one of the ways Method 6 would illustrate the interference condition is by displaying the endpoints of the loop-independent flow edge [i := 0] $\longrightarrow_f$ [(i <= 10)] in demoA together with the slice of demoB with respect to gmean := prod ** (1 / (i – 1)) in demoB.

demoA

```
program demo;
begin
   sum := 0;
   [[i := 0]];
   while [[(i <= 10)]] do
      sum := sum + i;
      i := i + 1
   od;
   amean := sum / i
end.
```

demoB

```
program demo;
begin
   sum := 0;
   <<prod := 1>>;
   <<i := 1>>;
   while <<(i <= 10)>> do
      sum := sum + i;
      <<prod := prod * i>>;
      <<i := i + 1>>
   od;
   <<gmean := prod ** (1 / (i – 1))>>
end.
```

Interference vertex (i <= 10) also affects statement prod := prod * i in demoB, so another one of the ways Method 6 would illustrate the interference condition is by displaying the endpoints of the edge [i := 0] $\longrightarrow_f$ [(i <= 10)] in demoA together with the slice of demoB with respect to prod := prod * i.

demoA

```
program demo;
begin
   sum := 0;
   [[i := 0]];
   while [[(i <= 10)]] do
      sum := sum + i;
      i := i + 1
   od;
   amean := sum / i
end.
```

demoB

```
program demo;
begin
   sum := 0;
   <<prod := 1>>;
   <<i := 1>>;
   while <<(i <= 10)>> do
      sum := sum + i;
      <<prod := prod * i>>;
      <<i := i + 1>>
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

Similarly, interference vertex i := i + 1 affects statements gmean := prod ** (1 / (i – 1)) and prod := prod * i in demoB, so Method 6 would also illustrate the interference condition of the example with the following two pairs of displays:

demoA

```
program demo;
begin
   sum := 0;
   [[i := 0]];
   while (i <= 10) do
      sum := sum + i;
      [[i := i + 1]]
   od;
   amean := sum / i
end.
```

demoB

```
program demo;
begin
   sum := 0;
   <<prod := 1>>;
   <<i := 1>>;
   while <<(i <= 10)>> do
      sum := sum + i;
      <<prod := prod * i>>;
      <<i := i + 1>>
   od;
   <<gmean := prod ** (1 / (i – 1))>>
end.
```

demoA

```
program demo;
begin
   sum := 0;
   [[i := 0]];
   while (i <= 10) do
      sum := sum + i;
      [[i := i + 1]]
   od;
   amean := sum / i
end.
```

demoB

```
program demo;
begin
   sum := 0;
   <<prod := 1>>;
   <<i := 1>>;
   while <<(i <= 10)>> do
      sum := sum + i;
      <<prod := prod * i>>;
      <<i := i + 1>>
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

Turning things around, Method 6 would also display in program B an edge (y, z) in the edge difference $(E(G_B / v) - E(G_A / v))$ while at the same time displaying in program A a slice G_A / w, where w is an affected point of B with respect to *Base* that is also affected by vertex v (*i.e.* $w \in (\text{AffectedBy}(\{ v \}, G_A) \cap AP_{A,Base})$).

Example. Interference vertex (i <= 10) affects statements sum := sum + i, i := i + 1, and amean := sum / i, which are all affected points of demoA with respect to demoBase, so Method 6 would illustrate the interference condition of the example with the following three pairs of displays:

demoA

```
program demo;
begin
   <<sum := 0>>;
   <<i := 0>>;
   while <<(i <= 10)>> do
      <<sum := sum + i>>;
      <<i := i + 1>>
   od;
   amean := sum / i
end.
```

demoB

```
program demo;
begin
   sum := 0;
   prod := 1;
   [[i := 1]];
   while [[(i <= 10)]] do
      sum := sum + i;
      prod := prod * i;
      i := i + 1
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

demoA

```
program demo;
begin
   sum := 0;
   <<i := 0>>;
   while <<(i <= 10)>> do
      sum := sum + i;
      <<i := i + 1>>
   od;
   amean := sum / i
end.
```

demoB

```
program demo;
begin
   sum := 0;
   prod := 1;
   [[i := 1]];
   while [[(i <= 10)]] do
      sum := sum + i;
      prod := prod * i;
      i := i + 1
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

demoA

```
program demo;
begin
   <<sum := 0>>;
   <<i := 0>>;
   while <<(i <= 10)>> do
      <<sum := sum + i>>;
      <<i := i + 1>>
   od;
   <<amean := sum / i>>
end.
```

demoB

```
program demo;
begin
   sum := 0;
   prod := 1;
   [[i := 1]];
   while [[(i <= 10)]] do
      sum := sum + i;
      prod := prod * i;
      i := i + 1
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

Finally, interference vertex i := i + 1 affects statements sum := sum + i, i := i + 1, and amean := sum / i in demoA, so Method 6 would also illustrate the interference condition of the example with the following three pairs of displays:

demoA

```
program demo;
begin
   <<sum := 0>>;
   <<i := 0>>;
   while <<(i <= 10)>> do
      <<sum := sum + i>>;
      <<i := i + 1>>
   od;
   amean := sum / i
end.
```

demoB

```
program demo;
begin
   sum := 0;
   prod := 1;
   [[i := 1]];
   while (i <= 10) do
      sum := sum + i;
      prod := prod * i;
      [[i := i + 1]]
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

demoA

```
program demo;
begin
   sum := 0;
   <<i := 0>>;
   while <<(i <= 10)>> do
      sum := sum + i;
      <<i := i + 1>>
   od;
   amean := sum / i
end.
```

demoB

```
program demo;
begin
   sum := 0;
   prod := 1;
   [[i := 1]];
   while (i <= 10) do
      sum := sum + i;
      prod := prod * i;
      [[i := i + 1]]
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

demoA

```
program demo;
begin
   <<sum := 0>>;
   <<i := 0>>;
   while <<(i <= 10)>> do
      <<sum := sum + i>>;
      <<i := i + 1>>
   od;
   <<amean := sum / i>>
end.
```

demoB

```
program demo;
begin
   sum := 0;
   prod := 1;
   [[i := 1]];
   while (i <= 10) do
      sum := sum + i;
      prod := prod * i;
      [[i := i + 1]]
   od;
   gmean := prod ** (1 / (i – 1))
end.
```

ACKNOWLEDGEMENTS

We thank Susan Horwitz for her helpful input as these ideas were developed.

REFERENCES

1. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," TR-690, Computer Sciences Department, University of Wisconsin, Madison, WI (March 1987).

2. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," pp. 133-145 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, (San Diego, CA, January 13-15, 1988), ACM, New York, NY (1988).

3. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," To appear in *ACM Trans. Program. Lang. Syst.*, (1989).

4. Reps, T. and Teitelbaum, T., "The Synthesizer Generator," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Pittsburgh, PA, Apr. 23-25, 1984), *ACM SIGPLAN Notices* **19**(5) pp. 42-48 (May 1984).

5. Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New York, NY (1988).

6. Reps, T. and Yang, W., "The semantics of program slicing," TR-777, Computer Sciences Department, University of Wisconsin, Madison, WI (June 1988).