IDENTIFYING THE SEMANTIC AND TEXTUAL DIFFERENCES BETWEEN TWO VERSIONS OF A PROGRAM

by

Susan Horwitz

Computer Sciences Technical Report #895

November 1989
Identifying the Semantic and Textual Differences Between Two Versions of a Program

SUSAN HORWITZ
University of Wisconsin – Madison

Text-based file comparators (e.g., the Unix utility diff) are very general tools that can be applied to arbitrary files. However, using such tools to compare programs can be unsatisfactory because their only notion of change is based on program text rather than program behavior. This paper describes a technique for comparing two versions of a program, determining which program components represent changes, and classifying each changed component as representing either a semantic or a textual change.

Key words and phrases: file comparison, file difference, language-based tools, program maintenance, semantic difference.

1. INTRODUCTION

A tool that detects and reports differences between versions of programs is of obvious utility in a software-development environment. Text-based tools, such as the Unix utility diff, have the advantage of being applicable to arbitrary files; however, using such tools to compare programs can be unsatisfactory because no distinction can be made between textual and semantic changes.

This paper describes a technique for comparing two programs, Old and New, determining which components of New represent changes from Old, and classifying each changed component as representing either a textual or a semantic change. It is, in general, undecidable to determine precisely the set of semantically changed components of New; thus, the technique described here computes a safe approximation to (i.e., possibly a superset of) this set. This computation is performed using a graph representation for programs and a partitioning operation on these graphs first introduced in [Yang89], and summarized in Section 2. The partitioning algorithm is currently limited to a language with scalar variables, assignment statements, conditional statements, while loops, and output statements. Because the partitioning algorithm is fundamental to the program-comparison algorithm described here, the program-comparison algorithm is also currently limited to the language described above. However, research is under way to expand the language; in particular, we are studying extensions for procedures and procedure calls, pointers, and arrays.

A precise definition of semantic change is given in Section 2; informally, a component c of New represents a semantic change either if there is no corresponding component of Old (because component c was added to Old to create New), or if a different sequence of values might be produced at c than at the corresponding component of Old. By “the sequence of values produced at c” we mean: if c is an assignment statement, the sequence of values assigned to the left-hand-side variable when the program is executed; if c is a predicate, the sequence of true-false values to which c evaluates when the program is executed; if c is an output statement, the sequence of values output when the program is executed.

This work was supported in part by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K, by the National Science Foundation under grant CCR-8958530, and by grants from Xerox, Eastman Kodak, and the Cray Research Foundation.

Author’s address: Computer Sciences Department, Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53706.
Example. Figure 1 shows a program *Old* and three different *New* programs; each *New* program is annotated to show its changes with respect to *Old*.

It is worthwhile to consider whether other approaches to program comparison could be used to detect the kinds of changes illustrated in Figure 1. In program *New*₁, the assignment “x := 2” is flagged as a semantic change because the value 2 is assigned to variable x whereas the corresponding component of *Old* assigns the value 1 to x. A text-based program comparator would also have flagged this as a changed component; however, the other changes flagged in *New*₁ would not have been detected by a text-based program comparator. These components represent semantic changes because they may use (directly or indirectly) the new value assigned to x.

The second and third semantic changes of program *New*₁ could have been detected by following def-use chains [Aho86] from the modified definition of x; however, program *New*₂ illustrates a situation in which following def-use chains leads to an erroneous detection of semantic change. In *New*₂, component “x := 0” is flagged as a semantic change because the sequence of values produced there is empty if variable P is true,¹ while the sequence of values produced at the corresponding component in *Old* is never empty (since the assignment is unconditional). Although “x := 0” represents a semantic change, the sequence of values produced at component “y := x” in *New*₂ is identical to the sequence of values produced at the corresponding component of *Old*; thus, “y := x” is not flagged as a change. Following def-use chains from “x := 0” would (incorrectly) identify both “y := x” and “output(y)” as semantic changes.

Finally, *New*₃ illustrates purely textual changes; again, following def-use chains from the changed component “y := a” would incorrectly identify “output(y)” as a semantic change.

In discussing the examples of Figure 1 we have talked about “corresponding components” in *Old* and the various *New* programs. How is this correspondence actually established? One possibility is to rely on the

```
<table>
<thead>
<tr>
<th>Old</th>
<th><em>New</em>₁</th>
<th><em>New</em>₂</th>
<th><em>New</em>₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 0</td>
<td>x := 0</td>
<td>if P then</td>
<td>a := 0</td>
</tr>
<tr>
<td>if P then</td>
<td>if P then</td>
<td>x := 1</td>
<td>TEXTUAL</td>
</tr>
<tr>
<td>x := 1</td>
<td>x := 2</td>
<td>else</td>
<td>if P then</td>
</tr>
<tr>
<td></td>
<td>SEMANTIC</td>
<td>x := 0</td>
<td>a := 1</td>
</tr>
<tr>
<td></td>
<td>fi</td>
<td>SEMANTIC</td>
<td>TEXTUAL</td>
</tr>
<tr>
<td>fi</td>
<td>fi</td>
<td>fi</td>
<td>fi</td>
</tr>
<tr>
<td>y := x</td>
<td>y := x</td>
<td>y := x</td>
<td>y := a</td>
</tr>
<tr>
<td>output(y)</td>
<td>output(y)</td>
<td>output(y)</td>
<td>TEXTUAL</td>
</tr>
<tr>
<td></td>
<td>SEMANTIC</td>
<td></td>
<td>output(y)</td>
</tr>
</tbody>
</table>
```

Figure 1. Program *Old* and three versions of *New*; each version of *New* is annotated to show its changes with respect to *Old*.

¹The language under consideration does not include explicit input statements. However, variables can be used before being defined; these variables’ values come from the initial state.
editing sequence used to create New from Old. For example, this correspondence could be established and maintained by the editor used to create New from Old as follows: Each component of Old has a unique tag; when a component is added, it is given a new tag, when a component is moved or modified it maintains its tag, when a component is deleted, its tag is never reused.

An algorithm for detecting the semantic and textual changes between Old and New, assuming editor-supplied tags, is given in Section 3.1; however, this approach has two important disadvantages:

1. A special editor that maintains tags is required.
2. The set of changes in New with respect to Old depends not only on the semantics of the two programs, but also on the particular editing sequence used to create New from Old. For example, it would be possible to use two different editing sequences to create programs New and New' from Old, such that the two new programs were identical, yet had different sets of changed components with respect to Old.

Section 3.2 considers how to determine semantic and textual changes between Old and New in the absence of editor-supplied tags; i.e., the problem of finding the correspondence between the components of Old and New is included as part of the program-comparison algorithm. A reasonable criterion for determining the correspondence is that it should minimize the difference between Old and New; however, we show that it is not satisfactory to define "difference between Old and New" as simply the number of semantically or textually changed components of New with respect to Old. Instead, we propose defining "difference between Old and New" as the number of semantically or textually changed components of New plus the number of new flow or control dependence edges in the graph representation of New (flow and control dependence edges are defined in Section 2). Finding a correspondence that minimizes the difference between Old and New according to this definition is shown to be NP-hard in the general case; a study of real programs is needed to determine how difficult the problem will be in practice.

2. PARTITIONING PROGRAM COMPONENTS ACCORDING TO THEIR BEHAVIORS

The program-comparison algorithm described in this paper relies on an algorithm for partitioning program components (in one or more programs) so that two components are in the same partition only if they have equivalent behaviors [Yang89]. The Partitioning Algorithm uses a graph representation of programs called a Program Representation Graph. This section summarizes the definitions of Program Representation Graphs and partitioning given in [Yang89].

2.1. The Program Representation Graph

Program Representation Graphs (PRGs) are currently defined only for programs in a limited language that includes scalar variables, assignment statements, conditional statements, while loops, and output statements.2

PRGs combine features of program dependence graphs [Kuck81,Ferrante87,Horwitz88] and static single assignment forms [Shapiro70,Alpern88,Cytron89,Rosen88]. A program's PRG is defined in terms of an augmented version of the program's control-flow graph. The standard control-flow graph includes a special Entry vertex and one vertex for each if or while predicate, each assignment statement, and each

---

2The language used in [Yang89] is actually slightly more restrictive, including only a limited kind of output statement called an end statement, which can appear only at the end of a program; however, it is clear that no problems are introduced by allowing general output statements.
output statement in the program. As in static single assignment forms, the control-flow graph is augmented by adding special \( \phi \) vertices so that each use of a variable in an assignment statement, an output statement, or a predicate is reached by exactly one definition.

1. For each variable \( x \) that is defined within either (or both) branches of an if statement and is live at the end of the if statement, a \( \phi_{if} \) vertex labeled \( \phi_{if}: x := x' \) is added to the control-flow graph immediately following the if statement. If there is more than one such vertex, their relative order is arbitrary.

2. For each variable \( x \) that is defined within a while loop, and is live immediately after the loop predicate (i.e., may be used before being redefined either inside the loop or after the loop), a \( \phi_{enter} \) vertex labeled \( \phi_{enter}: x := x' \) is added to the control-flow graph inside the loop, before the loop predicate. If there is more than one such vertex, their relative order is arbitrary.

3. For each variable \( x \) that is defined within a while loop and is live after the loop, a \( \phi_{exit} \) vertex labeled \( \phi_{exit}: x := x' \) is added to the control-flow graph immediately after the loop. If there is more than one such vertex, their relative order is arbitrary.

In addition, for each variable \( x \) that may be used before being defined (i.e., there is a \( x \)-definition clear path in the standard control-flow graph from the Entry vertex to a vertex that uses \( x \)), a vertex labeled \( x := Initial(x) \) is added to the control-flow graph after the Entry vertex. This vertex represents the assignment to \( x \) of a value from the initial state. If there is more than one such vertex, their relative order is arbitrary; however, they must appear sequentially, following the Entry vertex and preceding all other vertices in the control-flow graph.

Example. Figures 2(a) and 2(b) show a program and its augmented control-flow graph.

The vertices of a program's Program Representation Graph (PRG) are the same as the vertices in the augmented control-flow graph (an Entry vertex, one vertex for each predicate, each assignment statement, and each output statement, and for each Initial, \( \phi_{if} \), \( \phi_{enter} \), and \( \phi_{exit} \) vertex). The edges of the PRG represent control and flow dependences.

The source of a control dependence edge is always either the Entry vertex or a predicate vertex; control dependence edges are labeled either true or false. The intuitive meaning of a control dependence edge from vertex \( v \) to vertex \( w \) is that if the program component represented by vertex \( v \) is evaluated during program execution and its value matches the label on the edge, then, (assuming termination of all loops) the component represented by \( w \) will eventually execute. (By definition, the Entry vertex always evaluates to true.)

Algorithms for computing control dependences in languages with unrestricted control flow are given in [Ferrante87, Cytron89]. For the restricted language under consideration here, control dependence edges reflect the nesting structure of the program (i.e., there is an edge labeled true from the vertex that represents a while predicate to all vertices that represent statements inside the loop; there is an edge labeled true from the vertex that represents an if predicate to all vertices that represent statements in the true branch of the if, and an edge labeled false to all vertices that represent statements in the false branch; there is an edge labeled true from the Entry vertex to all vertices that represent statements or predicates that are not inside any while loop or if statement). In addition, there is a control dependence edge labeled true from every vertex that represents a while predicate to itself.

Flow dependence edges represent possible flow of values, i.e. there is a flow dependence edge from vertex \( v \) to vertex \( w \) if vertex \( v \) represents a program component that assigns a value to some variable \( x \), vertex \( w \) represents a component that uses the value of variable \( x \), and there is an \( x \)-definition clear path from \( v \) to \( w \) in the augmented control-flow graph.
```latex
x := 0
if P then
  x := 1
fi
y := x
output(y)
```

(a) 

```latex
p := Initial(P)
x := 0
if P
  x := 1
  \phi_U: x := x
  y := x
  output(y)
```

(b) 

```latex
p := Initial(P)
x := 0
if P
  \phi_U: x := x
  y := x
output(y)
```

(c) 

**Figure 2.** (a) A program; (b) its augmented control-flow graph; (c) its Program Representation Graph. In the Program Representation Graph, control dependence edges are shown using bold arrows and are unlabeled (in this example, all control dependence edges would be labeled true); data dependence edges are shown using arcs.

**Example.** Figure 2(c) shows the Program Representation Graph of the program of Figure 2(a). Control dependence edges are shown using bold arrows and are unlabeled (in this example, all control dependence edges would be labeled true); data dependence edges are shown using arcs.

2.2. The Partitioning Algorithm

The Partitioning Algorithm of [Yang89] can be applied to the Program Representation Graphs of one or more programs. The algorithm partitions the vertices of the graph(s) so that two vertices are in the same partition only if the program components that they represent have equivalent behaviors in the following sense:

**Definition** (equivalent behavior of program components). Two components $c_1$ and $c_2$ of (not necessarily distinct) programs $P_1$ and $P_2$ respectively, have equivalent behaviors iff all four of the following hold:

1. For all initial states $\sigma$ such that both $P_1$ and $P_2$ halt when executed on $\sigma$, the sequence of values produced at component $c_1$ when $P_1$ is executed on $\sigma$ is identical to the sequence of values produced at component $c_2$ when $P_2$ is executed on $\sigma$.

2. For all initial states $\sigma$ such that neither $P_1$ nor $P_2$ halts when executed on $\sigma$, either the sequence of values produced at component $c_1$ is an initial sub-sequence of the sequence of values produced at $c_2$ or vice versa.
(3) For all initial states $\sigma$ such that $P_1$ halts on $\sigma$ but $P_2$ fails to halt on $\sigma$, the sequence of values produced at $c_2$ is an initial sub-sequence of the sequence of values produced at $c_1$.

(4) For all initial states $\sigma$ such that $P_2$ halts on $\sigma$ but $P_1$ fails to halt on $\sigma$, the sequence of values produced at $c_1$ is an initial sub-sequence of the sequence of values produced at $c_2$.

By "the sequence of values produced at a component" we mean: for an assignment statement (including Initial statements and $\phi$ statements), the sequence of values assigned to the left-hand-side variable; for an output statement, the sequence of values output; and for a predicate, the sequence of boolean values to which the predicate evaluates.

The Partitioning Algorithm uses a technique (which we will call the Basic Partitioning Algorithm) adapted from [Alpern88, Aho74] that is based on an algorithm of [Hopcroft71] for minimizing a finite state machine. This technique finds the coarsest partition of a graph that is consistent with a given initial partition of the graph's vertices. The algorithm guarantees that two vertices $v$ and $v'$ are in the same class after partitioning if and only if they are in the same initial partition, and, for every predecessor $u$ of $v$, there is a corresponding predecessor $u'$ of $v'$ such that $u$ and $u'$ are in the same class after partitioning.

The Partitioning Algorithm operates in two passes. Both passes use the Basic Partitioning Algorithm, but apply it to different initial partitions, and make use of different sets of edges. The first pass creates an initial partition based on the operators that are used in the vertices; flow dependence edges are used by the Basic Partitioning Algorithm to refine this partition. The second pass starts with the final partition produced by the first pass; control dependence edges are used by the Basic Partitioning Algorithm to further refine this partition.

The time required by the Partitioning Algorithm is $O(N \log N)$, where $N$ is the size of the Program Representation Graph(s) (i.e., number of vertices + number of edges).

Example. Figure 3 illustrates partitioning using the programs from Figure 1. Figure 3 shows the partitions created by the Partitioning Algorithm: the initial partition, the refinement created by Pass 1, and the final partition. Note that the components labeled "$y := x$" from $Old$ and $New_2$ are in the same final partition (and thus have the same execution behavior) even though they are transitively flow dependent on components that are not in the same final partition (namely, the components labeled "$x := 0$" from $Old$ and $New_2$).

3. COMPUTING SEMANTIC AND TEXTUAL DIFFERENCES

This section presents three different algorithms to compute the semantic and textual differences between two versions of a program. All three algorithms operate on the programs' Program Representation Graphs; thus, in what follows, $New$ and $Old$ are Program Representation Graphs, and "program component" and "Program Representation Graph vertex" are used interchangeably.

Section 3.1 assumes that a special tag-maintaining editor is used to create program $New$ from program $Old$. Section 3.2 assumes that the correspondence between the components of $New$ and $Old$ must be computed; Sections 3.2.1 and 3.2.2 use different criteria for determining the best correspondence. In both cases the goal is to find a correspondence that minimizes the size of the change between $New$ and $Old$. However,
## Initial Partition

<table>
<thead>
<tr>
<th>Old</th>
<th>New 1</th>
<th>New 2</th>
<th>New 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ entry }</td>
<td>{ entry }</td>
<td>{ entry }</td>
<td>{ entry }</td>
</tr>
<tr>
<td>{ p := Init(p) }</td>
<td>{ p := Init(p) }</td>
<td>{ p := Init(p) }</td>
<td>{ p := Init(p) }</td>
</tr>
<tr>
<td>{ if \ p }</td>
<td>{ if \ p }</td>
<td>{ if \ p }</td>
<td>{ if \ p }</td>
</tr>
<tr>
<td>{ x := b }</td>
<td>{ x := b }</td>
<td>{ x := b }</td>
<td>{ x := b }</td>
</tr>
<tr>
<td>{ x := 1 }</td>
<td>{ x := 1 }</td>
<td>{ x := 0 }</td>
<td>{ x := 1 }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>After Pass 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old</td>
</tr>
<tr>
<td>{ entry }</td>
</tr>
<tr>
<td>{ p := Init(p) }</td>
</tr>
<tr>
<td>{ if \ p }</td>
</tr>
<tr>
<td>{ x := b }</td>
</tr>
<tr>
<td>{ x := 1 }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>After Pass 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old</td>
</tr>
<tr>
<td>{ entry }</td>
</tr>
<tr>
<td>{ p := Init(p) }</td>
</tr>
<tr>
<td>{ if \ p }</td>
</tr>
<tr>
<td>{ x := b }</td>
</tr>
<tr>
<td>{ x := 1 }</td>
</tr>
</tbody>
</table>

Figure 3. Partitioning Example. The partitions created by the Partitioning Algorithm for the programs of Figure 1.

in Section 3.2.1 "size of the change" is defined to be the number of semantically or textually changed
components of New, while in Section 3.2.2 "size of the change" is defined to be the number of semantically or textually changed components, plus the number of new flow or control dependence edges in New.

3.1. Component Correspondence is Maintained by the Editor

If program New is created from program Old using an editor that maintains tags on program components, then determining which components of New represent changes from Old and classifying each changed component as either a textual or semantic change is quite straightforward. A procedure called ComputeChanges that classifies the components of New is given below. The procedure first partitions programs Old and New and then considers each component c of New. If there is no component of Old with the same tag, then c was added to Old to create New, and thus represents a semantic change. Similarly, if there is a component of Old with the same tag, but the component is not in the same partition as c, then c represents a semantic change. If there is a component of Old with the same tag and in the same partition but with different text, then c represents a textual change.

procedure ComputeChanges( Old, New: Program Representation Graphs )
returns two sets of components of New, representing semantic and textual changes, respectively

declare semanticChange, textualChange: sets of program components

begin
apply the Partitioning Algorithm to Old and New
semanticChange := Ø
textualChange := Ø

for each component c of New do
if (there is no component of Old with the same tag as c) or
(the component of Old with the same tag as c is not in the same partition as c)
then insert c into semanticChange
else if the text of the component of Old that has the same tag as c ≠ the text of c
then insert c into textualChange
fi
fi
od
return( semanticChange, textualChange )
end

Procedure ComputeChanges can be illustrated by considering programs Old and New2 of Figure 1. Assume that program New2 was created from Old by moving the statement "x := 0" into the else branch of the if statement. In this case, for every component of New2 there is a component of Old with the same tag, and (as illustrated in Figure 3) for every component of New2 other than component "x := 0", the component of Old with the same tag is in the same final partition. Thus, the only component of New2 identified by procedure ComputeChanges as representing a change from Old is component "x := 0", which is identified as a semantic change.

3.2. Component Correspondence Must be Computed

In this section we consider how to compare programs Old and New assuming that program components are not tagged by the editor. Instead, the correspondence between the components of Old and New must be computed as part of the program-comparison algorithm. Our goal is to find a correspondence that minimizes the size of the change between Old and New. Sections 3.2.1 and 3.2.2 consider two different definitions of "the size of the change."
3.2.1. Size of change = the number of semantically or textually changed components of New

If we define the size of the change between Old and New as the number of semantically or textually changed components of New, then it is possible to define an efficient algorithm to find a correspondence that minimizes this size. A procedure called MatchAndComputeChanges that computes such a correspondence and simultaneously classifies the components of New with respect to Old is given below. The procedure first tries to match every component of New with a component of Old that is both semantically and textually equivalent. Next, the procedure considers all unmatched components of New, attempting to match them with unmatched components of Old that are semantically equivalent but textually different. These components of New are classified as textual changes. Components of New that remain unmatched are classified as semantic changes.

Applying procedure MatchAndComputeChanges to programs Old and New_2 of Figure 1 will produce the result pictured in Figure 1 even if the components of the two programs are not tagged by the editor. All components of New_2 other than "x := 0" will be matched with a component of Old that is both semantically and textually equivalent; component "x := 0" will be unmatched, and so will be classified as a semantic change.

Procedure MatchAndComputeChanges first partitions Old and New, then makes two passes through New matching and classifying its components. Assuming that it is possible to determine in constant time whether there is an unmatched component of Old in the same partition and with the same text as a given component, the procedure works as follows:

procedure MatchAndComputeChanges( Old, New: Program Dependence Graphs )
returns (1) a map from components of New to components of Old, and
(2) two sets of components of New, representing semantic and textual changes, respectively
declare map: a set of program component pairs
    semanticChange, textualChange: sets of program components
begin
    apply the Partitioning Algorithm to Old and New
    map := Ø; semanticChange := Ø; textualChange := Ø
    for each component c of New do
        if there is an unmatched component c' of Old that is in the same partition as c and has the same text
            then insert the pair (c, c') into map
                mark c "matched"
                mark c' "matched"
        fi
    od
    for each unmatched component c of New do
        if there exists an unmatched component c' of Old that is in the same partition as c
            then insert the pair (c, c') into map
                insert c into textualChange
                mark c' "matched"
            else insert c into semanticChange fi
    od
    return( map, semanticChange, textualChange )
end
component of New, the time required for matching and classifying is linear in the size of New; thus, the
time required for procedure MatchAndComputeChanges is dominated by the time required for partitioning.
The total time for procedure MatchAndComputeChanges is \( O(N \log N) \), where \( N \) is the sum of the sizes of
Old and New.

### 3.2.2. Size of change includes the number of new edges in New

Simply minimizing the number of semantically and textually changed components does not always produce
a satisfactory classification of the components of New; this is illustrated in Figure 4. Figure 4 shows pro-
grams Old and New, and four possible mappings from the components of New to the components of Old.
All four mappings induce the same (minimum) number of changed components of New with respect to Old,
yet there is something intuitively more satisfying about the first two mappings than the third and forth
mappings. The problem with the third and forth mappings is that they “separate” a use of variable \( x \) from the
corresponding definition of \( x \).

We can avoid choosing mapping three or mapping four of Figure 4 by redefining the “size of the change
between Old and New” to take into account PRG edges as well as vertices.

**Definition** (a correspondence between New and Old). A correspondence between New and Old is a 1-
to-1 partial function \( f \) from vertices of New to vertices of Old such that (1) for all vertices \( v \) of New, \( f(v) \) is
either a vertex of Old, or is the special value \( \bot \) (\( f(v) = \bot \) means that there is no vertex of Old that
corresponds to vertex \( v \) of New), and (2) if \( f(v) = v' \), then vertices \( v \) and \( v' \) are in the same final partition.

**Definition** (unmatched vertex). A vertex \( v \) of New is unmatched under correspondence \( f \) iff \( f(v) = \bot \).

**Definition** (unmatched edge). An edge \( v_1 \rightarrow v_2 \) of New is unmatched under correspondence \( f \) iff any of
the following hold: (1) \( f(v_1) = \bot \); (2) \( f(v_2) = \bot \); (3) there is no edge \( f(v_1) \rightarrow f(v_2) \) in Old of the same
type\(^4\) as the edge \( v_1 \rightarrow v_2 \).

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
<th>Mapping</th>
<th>Changed Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>[O1] x := 1</td>
<td>[N1] x := 1</td>
<td>{([N1]-[O1]), ([N2]-[O2])}</td>
<td>N3, N4</td>
</tr>
<tr>
<td>[O2] y := x</td>
<td>[N2] y := x</td>
<td>{([N3]-[O1]), ([N4]-[O2])}</td>
<td>N1, N2</td>
</tr>
<tr>
<td>[N3] x := 1</td>
<td></td>
<td>{([N1]-[O1]), ([N4]-[O2])}</td>
<td>N2, N3</td>
</tr>
<tr>
<td>[N4] y := x</td>
<td></td>
<td>{([N2]-[O2]), ([N3]-[O1])}</td>
<td>N1, N4</td>
</tr>
</tbody>
</table>

**Figure 4.** Programs Old and New, and four possible mappings from the components of New to the components of Old.
Each mapping induces a set of changed components of size 2; however, the first two mappings each induce only one
new data dependence, while the second two mappings each induce two new data dependences.

\(^4\)A precise definition of edge type can be found in [Yang89]. Roughly, two edges are of the same type if (1) they are both control-
dependence edges, or (2) they are both flow-dependence edges for the same operand of the target vertex. For example, vertices \( x := y + z \) and \( a := b + c \) both have two variable operands, so both have two incoming flow-dependence edges. The edge “carrying” the
definition of variable \( y \) can only match the edge carrying the definition of variable \( b \); it cannot match the edge carrying the definition of
variable \( c \).
Definition (size of change between Old and New). The size of the change between Old and New induced by correspondence \( f \) is: (the number of unmatched vertices \( v \) of New) + (the number of matched vertices \( v \) of New such that \( f(v) = v' \) and the text of \( v \) is not identical to the text of \( v' \)) + (the number of unmatched edges of New).

Figures 5(a), 5(b), and 5(c) give a procedure for computing a correspondence between New and Old that minimizes the size of the change between Old and New as defined above. However, since the problem of finding such a correspondence is NP-hard (as shown below) it is unlikely that an efficient procedure can be defined.

```
declare global bestSoFar: a correspondence between New and Old
declare global smallestChangeSoFar: integer

procedure Match(Old, New: Program Representation Graphs)
returns: a correspondence between New and Old that minimizes the size of the change between Old and New
    declare map: a correspondence between New and Old
    declare workingSet: a set of vertices of New
    begin
        apply the Partitioning Algorithm to Old and New
        map := \( \emptyset \)
        /* match all "no-choice" vertices of New */
        for each partition that includes exactly one vertex \( v \) of New and one vertex \( v' \) of Old do
            insert \( (v, v') \) into map
            mark \( v \) "matched"
            mark \( v' \) "matched"
        od

        /* put all remaining matchable vertices of New into the working set */
        workingSet := \( \emptyset \)
        for all unmatched vertices \( v \) of New such that \( \exists \) an unmatched vertex of Old in the same partition do
            insert \( v \) into workingSet
        od

        /* try all possible correspondences; keep track of the best one found */
        bestSoFar := \( \emptyset \)
        smallestChangeSoFar := \( \infty \)
        TryMatches(map, workingSet)

        /* the best correspondence has been saved in global variable bestSoFar */
        return( bestSoFar )
    end
```

Figure 5(a). Procedure Match finds a correspondence between New and Old that minimizes the difference between Old and New. Procedure Match calls procedure TryMatches, which is shown in Figure 5(b).
procedure TryMatches( map: a correspondence between New and Old,
                      workingSet: a set of vertices of New )
begin
  if workingSet = Ø
    then /* no more matchable vertices of New
         * compute the size of the change induced by the current correspondence;
         * save the current correspondence if its change size is smaller than the best so far
         */
      if ChangeSize( map ) < smallestChangeSoFar
        then bestSoFar := map
            smallestChangeSoFar := ChangeSize( map )
      fi
  else /* try all remaining possible matches */
    select and remove an arbitrary vertex v from workingSet
    let P be v's partition In
    remove v from P
    [L1]:
      if ( # of unmatched vertices of New in P ) ≥ ( # of unmatched vertices of Old in P )
      then /* must try correspondences in which v is unmatched, too */
        TryMatches( map, workingSet )
      fi
    [L2]:
      for each unmatched vertex v' of Old in partition P do
        insert (v, v') into map
        mark v' "matched"
        TryMatches( map, workingSet )
        remove (v, v') from map
        mark v' "unmatched"
      od
    /* put vertex v back into partition P and into workingSet so that it will be there next time TryMatches is called */
    add v to partition P
    insert v into workingSet
  fi
end

Figure S(b). If there are no more matchable vertices of New, Procedure TryMatches computes the size of the change between Old and New induced by the current correspondence. Otherwise, it tries all correspondences consistent with the given (incomplete) correspondence.
procedure ChangeSize( map: a correspondence between New and Old )
returns the size of the change between Old and New induced by the given correspondence

/*
* "size of change" = (# of unmatched vertices of New) +
* (# of vertices of New matched with textually different vertices of Old) +
* (# of unmatched edges of New)
*/

declare size: integer
begin
    size := (# of vertices of New) + (# of edges of New)
    for each vertex v of New do
        if (v, v') is in map
            then if text(v) = text(v') then size := size - 1 fi
                for each edge v → w in New do
                    if (w, w') is in map
                        then if (∃ edge v' → w' in Old) and (type(v → w) = type(v' → w'))
                            then size := size - 1
                        fi
                    fi
                fi
        od
    return( size )
end

Figure 5(c). Procedure ChangeSize computes the size of the change between Old and New induced by the given correspondence.

Procedure Match of Figure 5(a) matches all "no-choice" vertices of New, i.e., those vertices in partitions that include exactly one vertex of Old and one vertex of New. Match then puts all matchable vertices of New (those vertices of New that are unmatched and are in partitions with at least one unmatched vertex of Old) into a working set, and calls procedure TryMatches to try all correspondences that include the "no-choice" matchings performed so far.

To understand procedure TryMatches, consider what it does when the working set is empty, when the working set contains exactly one vertex, and when the working set contains more than one vertex.

The working set is empty.

When the working set is empty there are no partitions that include both an unmatched vertex of New and an unmatched vertex of Old; i.e., a complete correspondence has been defined. In this case, procedure TryMatches computes the size of the change induced by the current correspondence; the current correspondence and its change size are saved if it is the best correspondence found so far. (The size of the change induced by the current correspondence is computed by procedure ChangeSize, shown in Figure 5(c).)
The working set contains one vertex \( v \).

In this case, \( v \) is removed from the working set and from its partition \( P \). Now there are two subcases: (1) partition \( P \) contains no unmatched vertex of \( \text{Old} \); (2) partition \( P \) contains one or more unmatched vertices of \( \text{Old} \). In the first case, the correspondence is complete; the test at line [L1] will succeed (because both the number of unmatched vertices of \( \text{New} \) in \( P \) and the number of unmatched vertices of \( \text{Old} \) in \( P \) are zero), and a recursive call to \( \text{TryMatches} \) (with an empty working set) will be made. This recursive call will compute the cost of the current correspondence.

In the second case, the test at line [L1] will fail, and the \( \text{for} \) loop at line [L2] will be executed. Each time around the loop the current correspondence is completed by matching vertex \( v \) with a different unmatched vertex of \( \text{Old} \) in \( P \), and a recursive call to \( \text{TryMatches} \) (with an empty working set) is made.

The working set contains more than one vertex.

In this case, an arbitrary vertex \( v \) is selected and removed from the working set. The test at line [L1] serves two (similar) purposes. First, if there are no unmatched vertices of \( \text{Old} \) in \( v \)'s partition \( P \), the test will succeed, guaranteeing that the current correspondence will be completed with \( v \) unmatched (the \( \text{for} \) loop at line [L2] will not serve this purpose since it will execute zero times). Second, if, after removing \( v \) from \( P \) there are still at least as many unmatched vertices of \( \text{New} \) as unmatched vertices of \( \text{Old} \) left in \( P \), the test will succeed, and the recursive call to \( \text{TryMatches} \) will complete the current correspondence in all possible ways with \( v \) unmatched. The \( \text{for} \) loop at line [L2] will take care of completions in which \( v \) is matched with an available vertex of \( \text{Old} \).

The time requirements of procedure \( \text{TryMatches} \) can be analyzed as follows. Let \( M \) be 1 + the maximum number of unmatched vertices of \( \text{Old} \) in a partition with at least one unmatched vertex of \( \text{New} \). Given a working set of size 1, \( \text{TryMatches} \) will make at most \( M \) recursive calls, each with an empty working set, so \( T(1) \leq M \). Given a working set of size \( n \), \( \text{TryMatches} \) will make at most \( M \) recursive calls, each with a working set of size \( n-1 \), so \( T(n) \leq M \times T(n-1) \). Solving this equation we find that the time required for a call to \( \text{TryMatches} \) with a working set of size \( n \) is \( O(M^n) \).

The value of \( n \) for the original call to \( \text{TryMatches} \) made from procedure \( \text{Match} \) is the number of matchable vertices of \( \text{New} \) that remain after all no-choice matches are made. It remains to be seen how large this value, as well as the value of \( M \), are in practice. An (unrealistic) upper bound for the time required by \( \text{TryMatches} \) is \( O(O^N) \), where \( O \) is the number of vertices in \( \text{Old} \), and \( N \) is the number of vertices in \( \text{New} \).

**Finding a minimum change correspondence is NP-hard.**

In this section we prove that finding a correspondence between \( \text{New} \) and \( \text{Old} \) that minimizes the size of the change between \( \text{New} \) and \( \text{Old} \) is NP-hard (where "size of the change" is as defined above). We call this problem the "Minimum Correspondence" problem. We show that the Minimum Correspondence problem is NP-hard by showing that a related problem, the "k-Correspondence" problem, is NP-complete. An algorithm for k-Correspondence answers the question, "Is there a correspondence that induces a change between \( \text{Old} \) and \( \text{New} \) of size \( \leq k \)?". It is clear that a solution to the Minimum Correspondence problem provides a solution to the k-Correspondence problem; thus, if the k-Correspondence problem is NP-complete, the Minimum Correspondence problem is NP-hard.

To show that k-Correspondence is NP-complete, we must

1. show that k-Correspondence is in NP, and
2. show that a polynomial-time solution to k-Correspondence can be used to find a polynomial-time solution to a known NP-complete problem.
It is clear that k-Correspondence is in NP; a nondeterministic algorithm to solve the k-Correspondence problem partitions Old and New, then, for each vertex \( v \) of \( \text{New} \)'s PRG, matches \( v \) with a (nondeterministically chosen) unmatched vertex of \( \text{Old} \)'s PRG that is in the same partition. Finally, the size of the change induced by the resulting correspondence is computed; if this size is less than or equal to \( k \), the algorithm returns \text{true} \), otherwise it returns \text{false} \).

Next, we show that a polynomial-time solution to k-Correspondence can be used to find a polynomial-time solution to 3-CNF-Satisfiability (a known NP-complete problem). We show that, given a 3-CNF formula, we can produce (in polynomial time) Program Representation Graphs \( \text{Old} \) and \( \text{New} \), and an integer \( k \), such that there is a correspondence between \( \text{New} \) and \( \text{Old} \) that induces a change of size \( \leq k \) if the given 3-CNF formula is satisfiable.

The following terminology is used: A 3-CNF formula uses a set of \( \text{variables} \) \( x_1, x_2, \ldots, x_n \). The formula consists of the conjunction of a set of \( \text{clauses} \) \( c_1, c_2, \ldots, c_m \). Each clause \( c_j \) is of the form \( (t_{j1} \lor t_{j2} \lor t_{j3}) \), where each term \( t_{jk} \) is a barred or unbarred variable. For example:

\[
(x_1 \lor \overline{x}_1 \lor x_2) \land (x_2 \lor x_3 \lor x_4)
\]

\[\begin{align*}
3-\text{CNF Formula} & \quad \text{Set of Variables} & \quad \text{Terms} \\
(t_{11} = x_1) & \quad \{x_1, x_2, x_3, x_4\} & \quad t_{21} = \overline{x}_2 \\
(t_{12} = x_1) & \quad \{x_1, x_2, x_3, x_4\} & \quad t_{22} = x_3 \\
(t_{13} = x_2) & \quad \{x_1, x_2, x_3, x_4\} & \quad t_{23} = x_4
\end{align*}\]

Figure 6 shows the general form of the Program Representation Graphs \( \text{Old} \) and \( \text{New} \) produced from a given 3-CNF formula; Figure 7 shows the PRGs for the example 3-CNF formula given above. In Figure 6, the notation \( "<ij>" \) (used in both \( \text{Old} \) and \( \text{New} \)) means the index of the variable that appears in the \( i^{th} \) term of the \( j^{th} \) clause; thus, \( \text{"v<ij>"} \) is the identifier whose first character is \( \text{"v"} \) and whose remaining characters are the index of the variable that appears in the \( 2^{nd} \) term of the \( j^{th} \) clause. The notation \( "t_{jk}" \) (used in \( \text{New} \)) means the \text{value} of the term \( t_{jk} \); i.e., \( t_{jk} \) is a barred or unbarred variable.

To understand how a solution to the given 3-CNF-Satisfiability problem provides a solution to the corresponding Minimum Matching problem and \text{vice versa} , consider the equivalence classes produced by applying the Partitioning Algorithm to the programs \( \text{Old} \) and \( \text{New} \) illustrated in Figure 6. These classes are shown in Figure 8. The equivalence classes that contain three vertices (one from \( \text{Old} \) and two from \( \text{New} \)) “force” each variable \( x_i \) to be assigned a unique value. (Think of the vertex from \( \text{Old} \), “\( v_i := i \)” as the value \text{true}; that vertex can be matched with at most one of the two vertices from \( \text{New} \); either vertex “\( x_i := i \)” or vertex “\( \overline{x}_i := i \)”.) The former corresponds to assigning variable \( x_i \) the value \text{true}; the latter corresponds to assigning variable \( x_i \) the value \text{false}.

The equivalence classes that contain four vertices (one from \( \text{Old} \) and three from \( \text{New} \)) “choose” one term from each clause. Again, the \( \text{Old} \) vertex, “\( cj := v<1j1> + v<2j2> + v<3j3> + j \)” can be matched with at most one of the three vertices from \( \text{New} \), either “\( cj := t_1 + y<j2> + y<j3> + j \)” or “\( cj := t_2 + y<j1> + t_3 + j \)” or “\( cj := y<j1> + y<j2> + t_3 + j \)” or “\( cj := y<j1> + y<j2> + t_3 + j \)” or “\( cj := y<j1> + v<j2> + v<j3> + j \)” corresponds to choosing term \( t_{11} \); matching the second \( \text{New} \) vertex corresponds to choosing term \( t_{12} \); and matching the third \( \text{New} \) vertex corresponds to choosing term \( t_{13} \). The same number of \text{vertices} of \( \text{New} \) can be matched whether or not the given 3-CNF formula is satisfiable; however, if the formula is not satisfiable, then the number of matched \text{flow edges} of \( \text{New} \) will be insufficient to allow the size of the change to be \( \leq k \).
Old's PRG:

```
Entry
\[ \text{if } P \]
\[ v_i := i \quad c_j := v_{<j1>} + v_{<j2>} + v_{<j3>} + j \]

one \[ v_i := i \] for each variable \( x_i \)
one \[ c_j := v_{<j1>} + v_{<j2>} + v_{<j3>} + j \] for each clause \( c_j \)
```

New's PRG:

```
Entry
\[ \text{if } P \]
\[ y_i := i \]
\[ x_i := i \]
\[ c_j := t_{j1} + y_{<j2>} + y_{<j3>} + j \]
\[ c_j := y_{<j1>} + t_{j2} + y_{<j3>} + j \]
\[ c_j := y_{<j1>} + y_{<j2>} + t_{j3} + j \]

one \[ y_i := i \] for each variable \( x_i \)
one \[ x_i := i \] for each variable \( x_i \)
one \[ x_i := i \] for each variable \( x_i \)
one \[ c_j := t_{j1} + y_{<j2>} + y_{<j3>} + j \] for each clause \( c_j \)
one \[ c_j := y_{<j1>} + t_{j2} + y_{<j3>} + j \] for each clause \( c_j \)
one \[ c_j := y_{<j1>} + y_{<j2>} + t_{j3} + j \] for each clause \( c_j \)
```

Figure 6. The general form of PRGs Old and New built from a given 3-CNF formula. Flow edges are omitted. Control edges are shown unlabeled (these edges would all be labeled true). See the text for an explanation of notation like "\( v_{<j1>} \)" and \( t_{j1} \).
Figure 7. PRGs Old and New produced from 3-CNF formula \((x_1 \lor x_1 \lor x_2) \land (\overline{x_3} \lor x_3 \lor x_4)\). All flow edges and some control edges are omitted from New’s PRG. Control edges are shown unlabeled (these edges would all be labeled true).
Equivalence classes produced by the Partitioning Algorithm

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
<th>New</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Entry)</td>
<td>(Entry)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(if P)</td>
<td>(if P)</td>
<td>(y1 := 1)</td>
<td>(y2 := 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(yn := n)</td>
<td></td>
</tr>
<tr>
<td>(v1 := 1)</td>
<td>(x1 := 1)</td>
<td>(x1 := 1)</td>
<td></td>
</tr>
<tr>
<td>(v2 := 2)</td>
<td>(x2 := 2)</td>
<td>(x2 := 2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(vn := n)</td>
<td>(xn := n)</td>
</tr>
<tr>
<td>(c1 := v&lt;11&gt; + v&lt;12&gt; + v&lt;13&gt; + 1)</td>
<td>(c1 := t11 + y&lt;12&gt; + y&lt;13&gt; + 1)</td>
<td>(c1 := y&lt;11&gt; + t12 + y&lt;13&gt; + 1)</td>
<td>(c1 := y&lt;11&gt; + y&lt;12&gt; + t13 + 1)</td>
</tr>
<tr>
<td>(c1 := v&lt;21&gt; + v&lt;22&gt; + v&lt;23&gt; + 2)</td>
<td>(c1 := t21 + y&lt;22&gt; + y&lt;23&gt; + 2)</td>
<td>(c1 := y&lt;21&gt; + t22 + y&lt;23&gt; + 2)</td>
<td>(c1 := y&lt;21&gt; + y&lt;22&gt; + t23 + 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(cm := v&lt;m1&gt; + v&lt;m2&gt; + v&lt;m3&gt; + m)</td>
<td>(cm := t_m1 + y&lt;m2&gt; + y&lt;m3&gt; + m)</td>
<td>(cm := y&lt;m1&gt; + t_m2 + y&lt;m3&gt; + m)</td>
<td>(cm := y&lt;m1&gt; + y&lt;m2&gt; + t_m3 + m)</td>
</tr>
</tbody>
</table>

Figure 8. The equivalence classes produced by applying the Partitioning Algorithm to programs *Old* and *New* shown in Figure 6.

The value selected for *k* for a given 3-CNF formula is:

\[
((\text{# of vertices and edges in New}) - (\text{# of potentially unchanged vertices in New} + \text{# of matchable edges in New})) - ((3 + (6 \times \text{#-of-variables}) + (15 \times \text{#-of-clauses})) - (2 + (1 + \text{#-of-variables} + (2 \times \text{#-of-clauses}))))
\]

(5 \times \text{#-of-variables}) + (13 \times \text{#-of-clauses})

This number is explained below.

1. **Number of vertices and edges in New**
   The number of vertices in *New* is \(2 + (3 \times \text{# vars}) + (3 \times \text{# clauses}):\) the *Entry* vertex, the "if P" vertex, one "yi := i" vertex for each variable, one "xi := i" vertex for each variable, and one "\(\neg x_i := i\)" vertex for each variable; and three "cj := ..." vertices for each clause.
The number of control dependence edges in New is one less than the number of vertices (because every vertex other than the Entry vertex has exactly one incoming control dependence edge); thus, there are \(1 + (3 \times \# \text{ vars}) + (3 \times \# \text{ clauses})\) control dependence edges.

The number of flow dependence edges in New is \(9 \times \# \text{ clauses}\); the only vertices with incoming flow dependence edges are the "cj := ..." vertices; each such vertex uses three variables (so it has three incoming flow dependence edges), and there are three such vertices for each clause.

(2) Number of potentially unchanged vertices in New

The only vertices of New that can be matched with textually and semantically identical vertices of Old are the Entry vertex and the "if P" vertex. These vertices can be matched whether or not the 3-CNF formula is satisfiable.

(3) Number of matchable edges in New

Control dependence edges

A control dependence edge in New can be matched only if both endpoints are matched (however, the endpoints need not be matched with textually identical vertices of Old). There are \((1 + \#-\text{of-variables} + \#-\text{of-clauses})\) matchable control dependence edges. These edges, enumerated in the figure below, can be matched whether or not the 3-CNF formula is satisfiable.

```
Entry
   ↓
   if P
       ↓
       if P
           ↓
           xi := i
       or
           ↓
           xi := i

if P
  ↓
  or
  ↓
  cj := ...

1
1 for each variable
1 for each clause
```

Flow dependence edges

There are \((\#-\text{of-clauses})\) matchable flow dependence edges. These edges, enumerated in the figure below, can all be matched only if the 3-CNF formula is satisfiable.

```
t_1 := <j1>
   ↓
   cj := t_1 + y <j2> + y <j3> + j
or

t_2 := <j2>
   ↓
   cj := y <j1> + t_2 + y <j3> + j
or

t_3 := <j3>
   ↓
   cj := y <j1> + y <j2> + t_3 + j
```

1 for each clause

As stated in the discussion above on the value of \(k\), it is always possible to find a correspondence that matches two vertices of New with textually and semantically equivalent vertices of Old, and matches \((1 + \#-\text{of-variables} + \#-\text{of-clauses})\) control-dependence edges of New. The crux of the proof that the 3-CNF formula is satisfiable if there exists a correspondence between New and Old that induces a change of size \(\leq k\), is showing that \((\#-\text{of-clauses})\) flow-dependence edges can be matched iff the 3-CNF formula is satisfiable.
It is clear that (#-of-clauses) flow-dependence edges can be matched if the 3-CNF formula is satisfiable. In this case, for each variable $x_i$ with value true, the Old vertex "$v_i \leftarrow 1$" is matched with the New vertex "$x_i := 1$"; for each variable $x_i$ with value false, the Old vertex "$v_i \leftarrow 1$" is matched with the New vertex "$\overline{x}_i := 1$". For each clause $c_j$ with true term $t_{jk}$, the Old vertex "$c_j := v_j1 + v_j2 + v_j3 + 1$" is matched with the New vertex that "chooses" term $t_{jk}$. The flow-dependence edge from the New vertex that "assigns" true to term $t_{jk}$ to the New vertex for clause $c_j$ that "chooses" term $t_{jk}$ is matched because both of its endpoints are matched.

**Example.** Figure 9 uses the example 3-CNF formula $(x_1 \lor \overline{x}_1 \lor x_2) \land (x_2 \lor x_3 \lor x_4)$ to illustrate how a solution to the 3-CNF Satisfiability problem can be used to find a solution to the k-Correspondence problem. In this example, the number of vertices in New is 20, the number of edges in New is 37, the number of potentially unchanged vertices in New is 2, and the number of matchable edges in New is 9; thus the value of $k$ for this example is 46. The correspondence shown in Figure 9 matches two vertices of New with textually and semantically equivalent vertices of Old (namely, the Entry vertex and the "if $P$" vertex); nine edges of New (seven control-dependence edges and two flow-dependence edges) are matched. Therefore, this correspondence induces a change of size 46 (18 unmatched vertices, 6 matched but textually different vertices, 12 unmatched control-dependence edges, and 16 unmatched flow-dependence edges), which is the value of $k$.

It is also clear that the 3-CNF formula is satisfiable if (#-of-clauses) flow dependence edges can be matched. By construction, each subgraph that corresponds to a clause can contribute at most one matched flow-dependence edge; thus, if (#-of-clauses) flow-dependence edges can be matched, there must be exactly one such edge in each "clause" subgraph. Again by construction, the source of the matched flow-dependence edge is a vertex either of the form "$x_i := 1$", or of the form "$\overline{x}_i := 1$". Since for each variable $x_i$ at most one of these two vertices can be matched, this matching provides a satisfying truth value assignment for the variables of the formula.

4. RELATED WORK

Related work falls into two categories: techniques for computing textual differences, and techniques for computing semantic differences. The first category includes techniques for comparing strings [Sankoff72, Wagner74, Nakatsu82, Tichy84, Miller85] and techniques for comparing trees [Selkow77, Lu79, Tai79, Zhang89]. Although such work has a different goal than the technique described here, these textual differencing techniques might be useful in practice as a compromise between requiring editor-supplied tags and solving an NP-hard problem; i.e., one of these algorithms might be used to compute tags for program components. Once tags are available, the procedure ComputeChanges of Section 3.1 can be used to classify the components of New. In this case, no special editor is required, and tags are not a function of the particular edit sequence used to create program New from program Old; however, there is no guarantee that the size of the change between Old and New will be minimal in the sense of Section 3.2.2.

Program slicing [Weiser84, Ottenstein84, Horwitz88a] is a technique for identifying just those program components that might affect the values of a given set of variables at a given program point. Slicing is used by the program integration algorithm of [Horwitz89] to determine the semantic differences between two versions of a program. Program slicing could potentially be used in place of partitioning to identify sets of components of Old and New that have the same execution behavior. However, in the absence of tags, it is not clear whether this could be done efficiently since it would include determining isomorphism of slices as a subproblem. Furthermore, partitioning is superior to slicing in the sense that the equivalence classes determined by partitioning are supersets of the equivalence classes determined using slicing.
Figure 9. The matching of the vertices and edges of New and Old that corresponds to a solution for the 3-CNF formula \((x_1 \lor \bar{x}_1 \lor x_2) \land (x_2 \lor x_3 \lor x_4)\). In the illustration of the unchanged vertices and matched edges of New, only the vertices shown inside circles are matched with textually identical vertices of Old.

[Yang89]; for example, the components "y := x" and "output(y)" of program New_2 of Figure 1, and the components "y := a" and "output(y)" of program New_3 of Figure 1 would be classified as semantic changes if slicing were used in place of partitioning.
5. CONCLUSIONS

We have discussed three algorithms for comparing two versions of a program and identifying their semantic and textual differences. All three algorithms use the technique for partitioning programs introduced in [Yang89]. Although the partitioning technique is currently applicable only to a limited language, we believe that it can be extended to include many standard programming language constructs. Extensions to the partitioning algorithm translate directly into extensions to the program-comparison algorithms; thus, we believe that the algorithms described here will soon be applicable to a reasonable language, for example, Pascal without procedure parameters.

After extending the partitioning algorithm, we will be able to implement the three program-comparison algorithms to determine how well they work in practice. We will determine whether the third algorithm, which in theory should provide a better classification of changes than the second algorithm, does so in practice, and whether or not the NP-hard matching problem that it incorporates makes it unusable on real programs.

References

Aho74.

Aho86.

Alpen88.

Cytron89.

Ferrante87.

Hopcroft71.

Horwitz88.

Horwitz88a.
Horwitz, S., Reps, T., and Binkley, D., “Interprocedural slicing using dependence graphs,”

Horwitz89.


Kuck81.


Lu79.


Miller85.


Nakatsu82.


Ottenstein84.


Rosen88.


Sankoff72.


Selkow77.


Shapiro70.


Tai79.


Tichy84.