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

Susan Horwitz
University of Wisconsin–Madison

Abstract

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.

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, Kodak, and Cray.

Author's address: Computer Sciences Department, Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53706.

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

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 [Yan89], 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.

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.

A technique for determining the semantic differences between two versions of a program based on comparing program slices [Wei84,Ott84] is used by the program-integration algorithm of [Hor89a]. This technique can be adapted to detect the kinds of changes illustrated in programs New_1 and New_3. However, slice comparison is less precise than the partitioning technique described in this paper; for example, using slice comparison the components "y := x" and "output(y)" of New_2 would be identified as semantic changes. Section 4 provides a more detailed discussion of the slice-comparison technique, including more examples for which slice comparison is less precise than partitioning.

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 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:

Old	New_1	New_2	New_3
x := 0	x := 0	**if** P **then**	a := 0 ⇽ TEXTUAL
if P **then**	**if** P **then**	x := 1	**if** P **then**
x := 1	x := 2 ⇽ SEMANTIC	**else**	a := 1 ⇽ TEXTUAL
fi	**fi**	x := 0 ⇽ SEMANTIC	**fi**
y := x	y := x ⇽ SEMANTIC	**fi**	y := a ⇽ TEXTUAL
output(y)	output(y) ⇽ SEMANTIC	y := x	output(y)
		output(y)	

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

[1]The 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.

(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 [Yan89]. 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 [Yan89].

2.1. The Program Representation Graph

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

[2]The language used in [Yan89] 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.

PRGs combine features of program dependence graphs [Kuc81,Fer87,Hor89a] and static single assignment forms [Sha70,Alp88,Cyt89,Ros88]. 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 output statement in the program. As in static single assignment forms, the control-flow graph is augmented by adding special "ϕ vertices" so that each use of a variable in an assignment statement, an output statement, or a predicate is reached by exactly one definition.

One vertex labeled "ϕ_{if}: x := x" is added at the end of each *if* statement for each variable *x* that is defined within either (or both) branches of the *if* and is live at the end of the *if*; one vertex labeled "ϕ_{enter}: x := x" is added inside each *while* loop immediately before the loop predicate for each variable *x* that is defined within the *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); one vertex labeled "ϕ_{exit}: x := x" is added immediately after the loop for each variable *x* that is defined within the loop and is live after the loop. In addition, for each variable *x* that may be used before being defined, a vertex labeled "$x := Initial(x)$" is added at the beginning of the control-flow graph. 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*, ϕ_{if}, ϕ_{enter}, and ϕ_{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 the following: 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; however, if the value does not match the label on the edge, then the component represented by *w* may never execute. (By definition, the *Entry* vertex always evaluates to **true**.)

Algorithms for computing control dependences in languages with unrestricted control flow are given in [Fer87,Cyt89]. 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

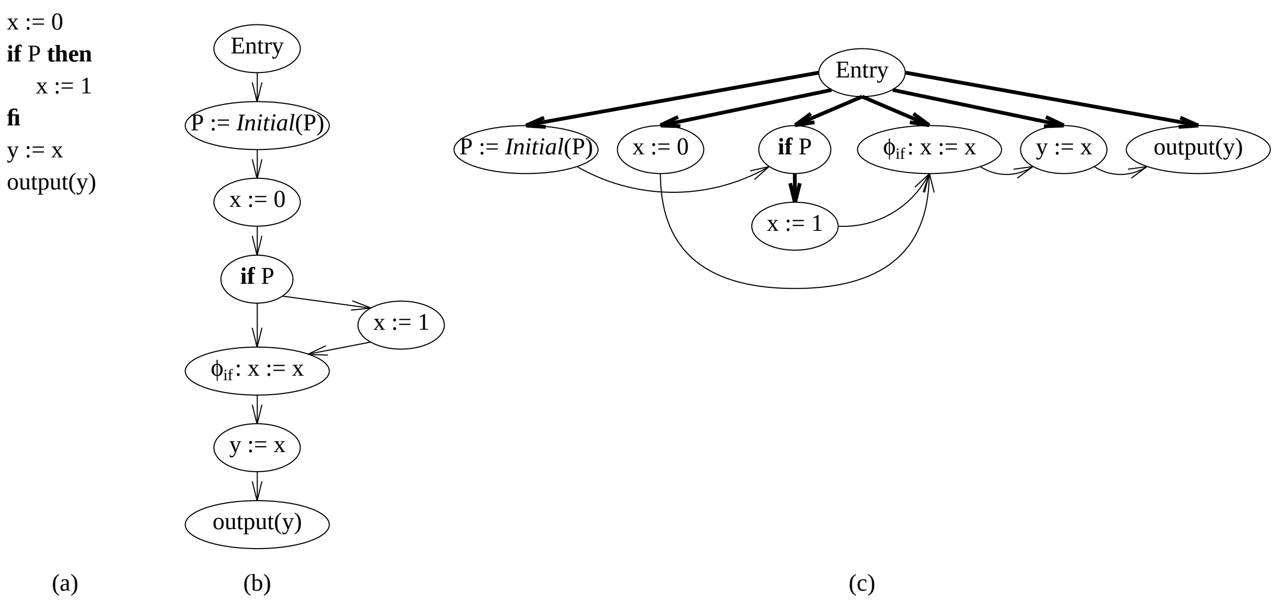

Figure 2. (a) A program; (b) its augmented control-flow graph; (c) its Program Representation Graph. In the Program Representation Graph, control dependence edges are shown using bold arrows and are unlabeled (in this example, all control dependence edges would be labeled **true**); data dependence edges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

edge labeled **true** from the *Entry* vertex to all vertices that represent statements 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.

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 [Yan89] 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 σ such that both P_1 and P_2 halt when executed on σ, the sequence of values produced at component c_1 when P_1 is executed on σ is identical to the sequence of values produced at component c_2 when P_2 is executed on σ.

(2) For all initial states σ such that neither P_1 nor P_2 halts when executed on σ, 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 σ such that P_1 halts on σ but P_2 fails to halt on σ, 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 σ such that P_2 halts on σ but P_1 fails to halt on σ, 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 ϕ 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 [Alp88,Aho74] that is based on an algorithm of [Hop71] 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 two of the partitions created by the Partitioning Algorithm: the initial partition 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 behaviors) 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

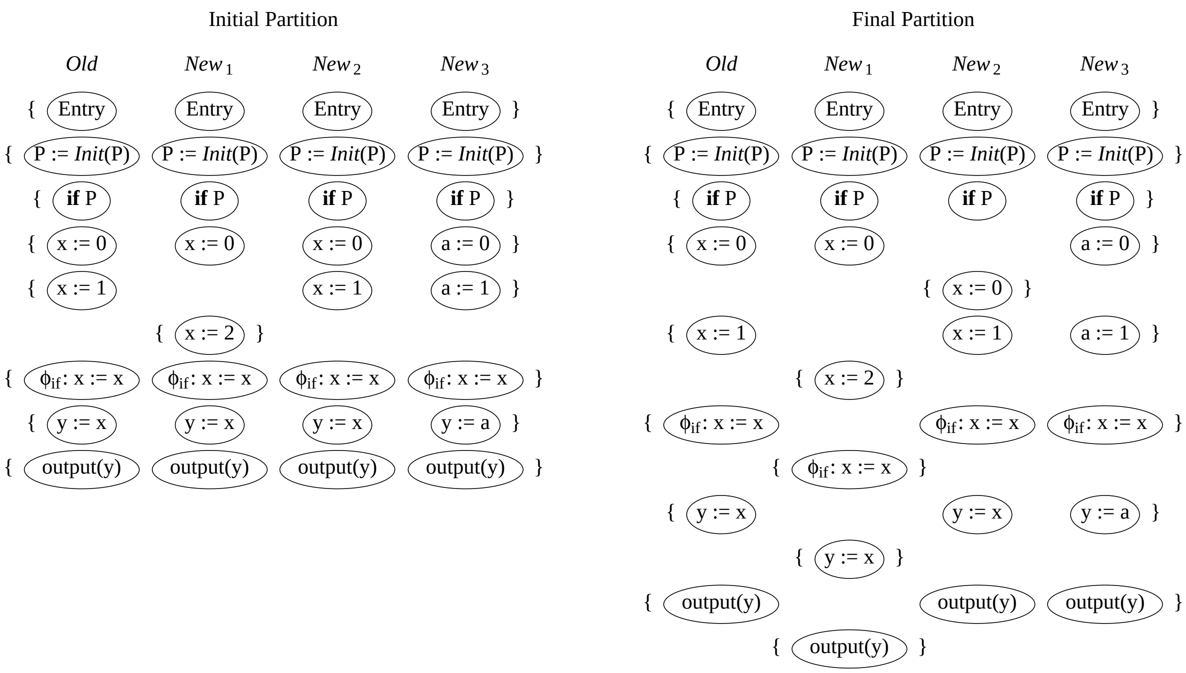

Figure 3. Partitioning Example. The partitions created by the Partitioning Algorithm for the programs of Figure 1.

between *New* and *Old*. However, 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 in Figure 4. Procedure ComputeChanges 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 can be illustrated by considering programs *Old* and New_2 of Figure 1. Assume that program New_2 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 New_2 there is a component of *Old* with the same tag, and (as illustrated in Figure 3) for every component of New_2 other than component "x := 0", the component of *Old* with the same tag is in the same final partition. Thus, the only component of New_2 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 in Figure 5. Procedure MatchAndComputeChanges 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

---

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 := ∅
    textualChange := ∅
    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

Figure 4. Procedure ComputeChanges classifies the components of *New* using editor-supplied tags..

```
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 := ∅
  semanticChange := ∅
  textualChange := ∅
  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 is an unmatched component c′ of Old that is in the same partition as c
    then insert the pair (c, c′) into map;  mark c "matched";  mark c′ "matched";  insert c into textualChange
    else insert c into semanticChange
    fi
  od
  return( map, semanticChange, textualChange )
end
```

Figure 5. Procedure MatchAndComputeChanges computes a correspondence between *New* and *Old* that minimizes the number of changed components of *New*.

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 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, which 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 6. Figure 6 shows programs *Old* and *New*, and four possible mappings from the components of *New* to the components of *Old*. All four mappings induce the same (minimal) 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 fourth mappings. The problem with the third and fourth 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 6 by redefining the "size of the change between

Old	*New*	Mapping	Changed Components
[O1] x := 1	[N1] x := 1	{([N1]-[O1]), ([N2]-[O2])}	N3, N4
[O2] y := x	[N2] y := x	{([N3]-[O1]), ([N4]-[O2])}	N1, N2
	[N3] x := 1	{([N1]-[O1]), ([N4]-[O2])}	N2, N3
	[N4] y := x	{([N2]-[O2]), ([N3]-[O1])}	N1, N4

Figure 6. 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.

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 $\perp$ ($f(v) = \perp$ 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 edge*). An edge $v_1 \rightarrow v_2$ of *New* is unmatched under the correspondence defined by f iff any of the following hold: (1) $f(v_1) = \perp$; (2) $f(v_2) = \perp$; (3) there is no edge $f(v_1) \rightarrow f(v_2)$ in *Old*.

Definition (*size of change between Old and New)*. The size of the change between *Old* and *New* induced by the correspondence defined by f is: (the number of vertices v of *New* such that $f(v) = \perp$) + (the number of 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*).

Figure 7 gives 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 [Hor89b] it is unlikely that an *efficient* procedure can be defined.

The procedure of Figure 7 works as follows. First, all "no-choice" vertices of *New* (*i.e.*, those vertices in partitions that include exactly one vertex of *Old* and one vertex of *New*) are matched with the (single) vertex of *Old* that is semantically equivalent. This is accomplished by procedure Match. Next, a backtracking scheme is used to try all possible matchings of the remaining vertices of *New* with the remaining vertices of *Old*. Each time a complete correspondence is defined, its cost is computed, and if its cost is the lowest found so far, the correspondence is saved. This backtracking is performed by procedure TryMatches, which is called from Match with an initial working set containing all *matchable* vertices of *New* (those vertices of *New* that are unmatched and are in partitions with at least one unmatched vertex of *Old*).

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. (Code for function ChangeSize has been omitted. This function computes the size of the change induced by the current correspondence, which is the number of unmatched vertices of *New* plus the number of vertices of *New* matched with textually different vertices of *Old* plus the number of unmatched edges of *New*.)

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 *Old*; (2) partition P contains one or more unmatched vertices of *Old*. In the first case, the correspondence is complete; the test at line [L1] will succeed (because both the number of unmatched vertices of *New* in P and the number of unmatched vertices of *Old* in P are zero), and a recursive call to 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 *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 *Old* in P, and a recursive call to 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 *Old* in v's partition P, the test will succeed, guaranteeing that the current correspondence will be completed with v unmatched (the *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 *New* as unmatched vertices of *Old* left in P, the test will succeed, and the recursive call to TryMatches will complete the current correspondence in all possible ways with v unmatched. The *for* loop at line [L2] will take care of completions in which v is matched with an available vertex of *Old*.

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

The value of n for the original call to TryMatches made from procedure Match is the number of matchable vertices of *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 TryMatches is $O(O^N)$, where O is the number of vertices in *Old*, and N is the number of vertices in *New*.

```
declare global bestSoFar: a correspondence between New and Old
                    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
              workingSet: a set of vertices of New
begin
   apply the Partitioning Algorithm to Old and New
   map := ∅
   /* 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 := ∅
       for all unmatched vertices v of New such that ∃ 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 := ∅; smallestChangeSoFar := ∞; TryMatches(map, workingSet)
   /* the best correspondence has been saved in global variable bestSoFar */
      return( bestSoFar )
end


procedure TryMatches( map: a correspondence between New and Old; workingSet: a set of vertices of New )
begin
   if workingSet = ∅
   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
         ni
   fi
end
```

Figure 7. Procedure Match finds a correspondence between *New* and *Old* that minimizes the difference between *Old* and *New*. Procedure Match first matches all “no-choice” vertices of *New* and then calls procedure TryMatches. 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 trys all correspondences consistent with the given (incomplete) correspondence.

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 [San72,Wag74,Nak82,Tic84,Mil85] and techniques for comparing trees [Sel77,Lu79,Tai79,Zha89]. 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.

As mentioned in Section 1, an important part of the program-integration algorithm of [Hor89a] is the identification of the changed computations of a program variant with respect to the original program. The technique used by that algorithm involves comparing program slices [Wei84,Ott84]. (The slice of a program with respect to a given component *c* is the set of program components that might affect the values of the variables used at component *c*.)

Slice comparison could be used in place of the Partitioning Algorithm to partition the components of programs *Old* and *New*; any of the three techniques for matching components of *Old* and *New* discussed in Section 3 could then be applied. Using this approach, a component of *New* is placed in the same partition as all components of *Old* and all other components of *New* that have identical slices.

To compare partitioning using the Partitioning Algorithm to partitioning using slice comparison we must consider: (1) the times required for each of the two techniques, and (2) the accuracy of the partitions computed by each of the two techniques.

Slice equality for a pair of program components can be determined in time linear in the size of the two slices; *i.e.*, given components c_1 and c_2, it is possible to determine whether the slices with respect to c_1 and c_2 are equal in time linear in the number of vertices and edges in the two slices [Hor91]. Given this result, a straightforward technique for partitioning programs *Old* and *New* using slice comparison is the following:

WorkingSet := (vertices of *New* $\cup$ vertices of *Old*)
while WorkingSet $\neq \emptyset$ **do**
  create a new, empty partition class *P*
  select and remove a vertex *v* from WorkingSet
  insert *v* into *P*
  for all vertices *u* in WorkingSet **do**
    if slice(*v*) = slice(*u*) **then**
      remove *u* from WorkingSet
      insert *u* into *P*
    fi
  od
od

This technique requires time $O(N^3)$, where N is the sum of the sizes of *Old* and *New*. An $O(N^2)$ algorithm for partitioning using slice comparison is described in [Hor91]; the better time bound is achieved through the use of structure sharing.

Next we consider how the partitions produced by the Partitioning Algorithm compare to those produced using slice comparison. If two slices are considered to be equal only if they have both identical structure and identical *text*, then partitioning using slice comparison produces partitions that are subsets of the partitions produced using the Partitioning Algorithm, and it is not possible to use these partitions to differentiate between textual and semantic changes. For example, components "x := 2", "y := x", and "output(y)" of program New_1 of Figure 1, as well as components "a := 0", "a := 1", "y := a", and "output(y)" of program New_3 would all be identified as *changed*, with no distinction made between the semantic changes of New_1 and the purely textual changes of New_3.

An algorithm that identifies as equal slices that are structurally identical, and textually identical up to variable renaming is given in [Hor91]. In this case, the partitions for programs *Old*, New_1, and New_3 produced using slice comparison would be the same as the partitions produced using the Partitioning Algorithm (and therefore the same components of New_1 and New_3 would be identified as semantic and textual changes). However, in general, the partitions produced using slice comparison would be subsets of the partitions produced using the Partitioning Algorithm. This is illustrated in Figure 8, which shows an *Old* program and three different *New* programs; components of the *New* programs that are semantically equivalent to the (obvious) corresponding component of *Old* (and that would be placed in the same partitions as the corresponding components of *Old* by the Partitioning Algorithm) but whose slices differ from the slices of the corresponding components of *Old* are flagged with arrows. The three examples illustrated in Figure 8 can be characterized as follows: (1) the component of *Old* uses a literal, and the corresponding component of New_4 uses a variable that has been assigned the literal's value; (2) the component of *Old* uses a variable x, and the corresponding component of New_5 uses a different variable that has been assigned x's value; (3) the components of *Old* and New_6 use values assigned using structurally different but semantically equivalent con-

Old	*New* $_4$	*New* $_5$	*New* $_6$
rad := 2	PI := 3.14	rad := 2	**if** DEBUG **then**
if DEBUG **then**	rad := 2	**if** DEBUG **then**	rad := 4
rad := 4	**if** DEBUG **then**	rad := 4	**else**
fi	rad := 4	**fi**	rad := 2
area := 3.14*(rad**2)	**fi**	area := 3.14*(rad**2)	**fi**
vol := height*area	area := PI*(rad**2) ⟵	tmp := area	area := 3.14*(rad**2) ⟵
	vol := height*area ⟵	vol := height*tmp ⟵	vol := height*area ⟵

Figure 8. Examples for which Yang et al's partitioning algorithm is superior to partitioning using slice comparison. Statements flagged with arrows are semantically equivalent to the corresponding statements in *Old*, but have different slices than the corresponding statements in *Old*.

structs involving conditional statements.

To summarize: slice comparison could be used in place of the Partitioning Algorithm to identify semantically equivalent components of *Old* and *New*. The time required for partitioning using slice comparison is $O(N^2)$ while the time required for partitioning using the Partitioning Algorithm is $O(N \log N)$; the partitions computed using slice comparison would be subsets of the partitions computed using the Partitioning Algorithm. It remains to be seen how the two techniques compare in practice.

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 [Yan89]. 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.
Aho, A., Hopcroft, J.E., and Ullman, J., *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA (1974).

Aho86.
Aho, A., Sethi, R., and Ullman, J., *Compilers: Principles, Techniques and Tools,* Addison-Wesley, Reading, MA (1986).

Alp88.
Alpern, B., Wegman, M.N., and Zadeck, F.K., "Detecting equality of variables in programs," pp. 1-11 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), (1988).

Cyt89.
Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, K., "An efficient method of computing static single assignment form," pp. 25-35 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), (1989).

Fer87.
Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* **9**(3) pp. 319-349 (July 1987).

Hop71.
Hopcroft, J.E., "An *n* log *n* algorithm for minimizing the states of a finite automaton," *The Theory of Machines and Computations,* pp. 189-196 (1971).

Hor89a.
Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Transactions on Programming Languages and Systems* **11**(3) pp. 345-387 (July 1989).

Hor89b.
Horwitz, S., "Identifying the semantic and textual differences between two versions of a program," Technical Report 895, Department of Computer Sciences, University of Wisconsin—Madison (November 1989).

Hor91.
Horwitz, S. and Reps, T., "Efficient comparison of program slices," *Acta Informatica* **28** pp. 713-732 (1991).

Kuc81.
Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and

Wolfe, M., “Dependence graphs and compiler optimizations,” pp. 207-218 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages,* (Williamsburg, VA, January 26-28, 1981), (1981).

Lu79.
Lu, S.Y., “A tree-to-tree distance and its application to cluster analysis,” *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-1**(2) pp. 219-224 (April 1979).

Mil85.
Miller, W. and Myers, E.W., “A file comparison program,” *Software − Practice & Experience* **15**(11) pp. 1025-1040 (November 1985).

Nak82.
Nakatsu, N., Kambayashi, Y., and Yajima, S., “A longest common subsequence algorithm suitable for similar text strings,” *Acta Informatica* **18** pp. 171-179 (1982). (as cited in [Miller85])

Ott84.
Ottenstein, K.J. and Ottenstein, L.M., “The program dependence graph in a software development environment,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, (Pittsburgh, PA, April 23-25, 1984), ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).

Ros88.
Rosen, B., Wegman, M.N., and Zadeck, F.K., “Global value numbers and redundant computations,” pp. 12-27 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), (1988).

San72.
Sankoff, D., “Matching sequences under deletion/insertion constraints,” *Proc. Nat. Acad. Sci.* **69**(1) pp. 4-6 (January 1972).

Sel77.
Selkow, S.M., “The tree-to-tree editing problem,” *Information Processing Letters* **6**(6) pp. 184-186 (December 1977).

Sha70.
Shapiro, R. M. and Saint, H., “The representation of algorithms,” Technical Reprot CA-7002-1432, Massachusetts Computer Associates (February 1970).

Tai79.
Tai, K.C., “The tree-to-tree correction problem,” *JACM* **26**(3) pp. 422-433 (July 1979).

Tic84.
Tichy, W., “The string-to-string correction problem with block moves,” *ACM Transactions on Computer Systems* **2**(4) pp. 309-321 (November 1984).

Wag74.
Wagner, R.A. and Fischer, M.J., “The string-to-string correction problem,” *JACM* **21**(1) pp. 168-173 (January 1974).

Wei84.
Weiser, M., “Program slicing,” *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).

Yan89.
Yang, W., Horwitz, S., and Reps, T., “Detecting program components with equivalent behaviors,” Technical Report 840, Department of Computer Sciences, University of Wisconsin, Madison, WI (April 1989).

Zha89.
Zhang, K. and Shasha, D., “Simple fast algorithms for the editing distance between trees and related problems,” *to appear in SIAM Journal of Computing,* (1989).