DEMONSTRATION OF A PROTOTYPE TOOL FOR PROGRAM INTEGRATION

THOMAS REPS
University of Wisconsin–Madison

This paper illustrates a sample session with a preliminary implementation of a program-integration tool. The tool has been embedded in a program editor created using the Synthesizer Generator, a meta-system for creating interactive, language-based program development systems. Information maintained by the editor front-end is used to construct dependence graphs. An integration command added to the editor invokes the integration algorithm on the dependence graphs, reports whether the variant programs interfere, and, if there is no interference, builds the integrated program.

It should be noted that the integration capabilities of the tool are severely limited; in particular, the tool can only handle programs written in a simple language in which the only statements are assignment statements, conditional statements, and while-loops.

1. INTRODUCTION

Our concern is the design and implementation of interactive environments for computer programming; our current goal is the development of a language-based tool for *integrating program variants*. By this, we mean a tool for automatically combining several related, but different variants of a base program, or determining that they incorporate interfering changes.

The need to integrate several versions of a program into a common one arises frequently, but it is a tedious and time consuming task to integrate programs by hand. One of the ways in which this situation arises is when a base version of a system is enhanced along different lines, either by users or maintainers, thereby creating several related versions with slightly different features. If one wishes to create a new version that incorporates several of the enhancements simultaneously, one has to check for conflicts in the implementations of the different versions and then merge them to create an integrated version that combines their separate features.

Recently, in collaboration with S. Horwitz and J. Prins, we developed a radically different approach to building an automatic program-integration tool. The cornerstone of the approach is a new algorithm that takes as input three programs *A*, *B*, and *Base*, where *A* and *B* are two variants of *Base*; whenever the changes made to *Base* to create *A* and *B* do not "interfere" in a certain sense, the algorithm produces a program *M* that integrates *A* and *B*. The method is based on the assumption that any change in the *behavior*, rather than the *text*, of *Base*'s variants is significant and must be preserved in *M*. Although it is undecidable to determine whether a program modification actually leads to such a difference, it is possible to determine a safe approximation by comparing each of the variants with *Base*. To determine this information, the integration algorithm employs a program representation that is similar (although not identical) to the *dependence graphs* that have been used previously in vectorizing and parallelizing compilers. The algorithm also make use of the notion of a *program slice* to find just those statements of a program that determine the values of potentially affected variables. (For details of the integration algorithm, see [2].)

A preliminary implementation of a program-integration tool that uses this technique has been embedded in a program editor created using the Synthesizer Generator, a meta-system for creating interactive, language-based program development systems [4]. Information maintained by the editor front-end is used to construct dependence graphs. An integration command added to the editor invokes the integration algorithm on the dependence graphs, reports whether the variant programs interfere, and, if there is no interference, builds the integrated program. (It should be noted, however, that the integration capabilities of the tool are severely limited; in particular, the tool can only handle programs written in a simple language in which the only statements are assignment statements, conditional statements, and while-loops.)

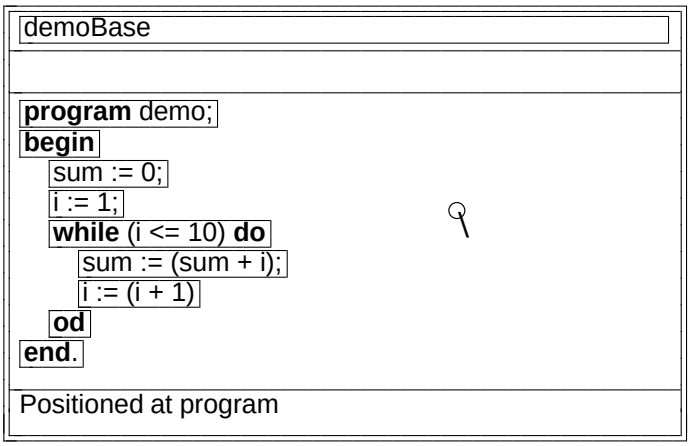
This paper demonstrates the capabilities of the prototype program-integration tool by presenting a sample session with it. The session demonstrates how dependences between program elements may be illustrated by program slicing, how new programs may be created through program integration, and how conflicts that arise in an interfering integration may be illustrated.

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

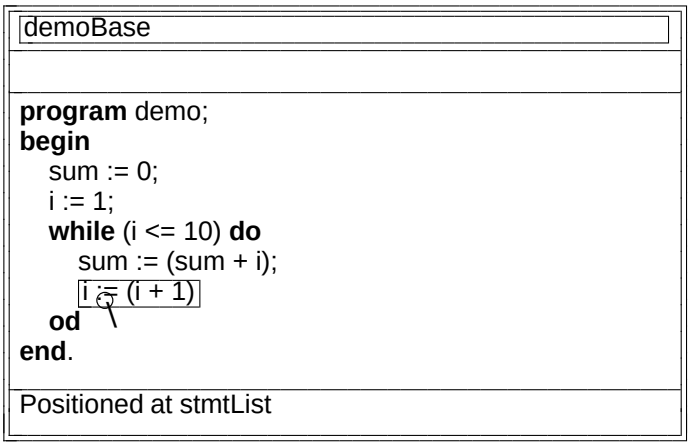
Author's address: Computer Sciences Department, University of Wisconsin – Madison, 1210 W. Dayton St., Madison, WI 53706.

2. PROGRAM SLICING AND PROGRAM INTEGRATION

This section illustrates the program-integration tool's operations of program slicing and program integration. We begin the session by invoking the editor on buffer demoBase, which contains a program to sum the integers from 1 to 10, as shown below:



To illustrate slicing, we first select statement i := (i + 1) by pointing the locator (indicated by) at any of the characters of i := (i + 1) and invoking the **select** command by clicking the left mouse button.



The highlighted region changes to indicate the extent of the new selection, and the help pane is updated to provide information about the currently selected component. The new selection is a stmtList, a list of statements that, for the moment, consists of the single statement i := (i + 1).

We now invoke the **slice** command (by selecting it from a menu of commands) to display all program elements—statements or predicates—that can affect the values of variables defined or used within the current selection. The **slice** command changes the display to indicate which program elements can affect the value of variable i at i := (i + 1); elements of the slice are displayed below in an italic typeface that is slightly larger than the typeface used for the other elements of the program; on a color monitor, the program elements that are part of the slice are displayed in a contrasting color.

demoBase

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

Positioned at stmtList

We now introduce buffer demoA, which contains a version of the program in buffer demoBase. This program, created by editing a *copy* of demoBase, is just like demoBase except for the additional statement at the end of the program, amean := (sum / (i – 1)), which computes the arithmetic mean.

demoA

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

Positioned at program

When the program in buffer demoA is sliced with respect to statement i := (i + 1), the slice consists of the same components that appeared in the slice of the program in buffer demoBase.

demoA

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

Positioned at stmtList

demoBase

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

Positioned at stmtList

Because the two slices are equal, we know that the two programs have the same execution behavior with respect to statement i := (i + 1); that is, variable i takes on the same sequence of values in both programs [5].

Now consider the slice of demoA with respect to statement amean := (sum / (i – 1)).

demoA

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

Positioned at stmtList

This slice contains the additional elements amean := (sum / (i – 1)) (the point with respect to which the slice is taken), sum := (sum + i) (because of the flow dependence from sum := (sum + i) to amean := (sum / (i – 1))), and sum := 0 (because of the flow dependence from sum := 0 to sum := (sum + i).

We now introduce buffer demoB, which contains a second version of the program in buffer demoBase. The program in demoB incorporates the computation of the geometric mean (but not the arithmetic mean, which was introduced solely in demoA).

demoB

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

Positioned at program

The geometric-mean computation introduces three new assignment statements: prod := 1 (to initialize the running product), prod := (prod * i) (to compute the running product), and gmean := (prod ** (1 / (i – 1))) (to compute the geometric mean as a function of prod and i).

Let us now slice the program in buffer demoB with respect to statement gmean := (prod ** (1 / (i – 1))).

demoB

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

Positioned at stmtList

Note that the slice includes neither sum := 0 nor sum := (sum + i) because neither can affect the values of variables prod, i, or gmean at gmean := (prod ** (1 / (i – 1))).

Let us now integrate buffers demoA and demoB with respect to buffer demoBase by invoking the **integrate** command. It is necessary to supply parameters to this command to specify the buffer that contains the base program, the buffers that contain the two variants, and the buffer in which the integrated program is to be placed. Thus, the **integrate** command calls up a parameter form as soon as the command is invoked.

Integration Form

Base program: <buffer-name>
Variant A: <buffer-name>
Variant B: <buffer-name>
Integrated program: INTEGRATED

A parameter form is a structured object, and parameters are entered by editing the form like any other buffer. By default, the field of the integration form specifying the name of the buffer into which the integrated program is to be placed is filled in with INTEGRATED.

Integration Form

Base program: demoBase
Variant A: demoA
Variant B: demoB
Integrated program: INTEGRATED

Once the form is filled in with appropriate parameters—in this case demoBase as the base program, demoA and demoB as the two variants—the rest of the command is processed by invoking the command **start-command**.

The integration of demoBase, demoA, and demoB succeeds and the program that results is placed in buffer INTEGRATED.

INTEGRATED

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

Positioned at program

The execution behavior of the program in buffer INTEGRATED combines the behaviors of the programs in demoBase, demoA, and demoB. For example, because the slice of the program in INTEGRATED with respect to statement gmean := (prod ** (1 / (i – 1))) is the same as the corresponding slice of the program in demoB, the final value computed for variable gmean by INTEGRATED is the same as that computed by demoB.

INTEGRATED

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

Positioned at stmtList

demoB

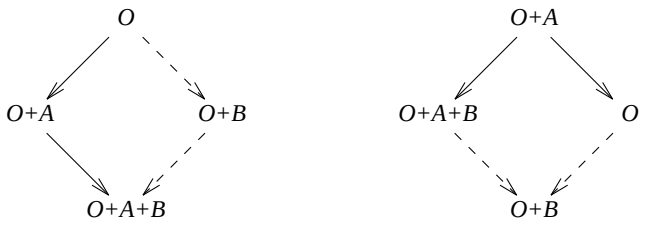
```
program demo;
begin
    sum := 0;
    prod := 1;
    i := 1;
    while (i <= 10) do
        sum := (sum + i);
        prod := (prod * i);
        i := (i + 1)
    od;
    gmean := (prod ** (1 / (i – 1)))
end.
```

Positioned at stmtList

Note that not only are statements sum := 0 and sum := (sum + i) left out of the slice of INTEGRATED (as is also the case in the slice of demoB), but statement amean := (sum / (i – 1)) is left out as well.

3. SEPARATING CONSECUTIVE PROGRAM MODIFICATIONS

Another application of program integration permits separating consecutive edits on the same program into individual edits on the original base program. For example, consider the case of two consecutive edits to a base program O; let $O+A$ be the result of the first modification to O and let $O+A+B$ be the result of the modification to $O+A$. Now suppose we want to create a program $O+B$ that includes the second modification but not the first. This is represented by situation (a) in the following diagram:



(a) (b)

Under certain circumstances, the development-history tree can be re-rooted so that $O+A$ is the root; the diagram is turned on its side and becomes a program-integration problem (situation (b)). The base program is now $O+A$, and the two variants of $O+A$ are O and $O+A+B$. Instead of treating the differences between O and $O+A$ as changes that were made to O to create $O+A$, they are now treated as changes made to $O+A$ to create O. For example, when O is the base program, a statement s that occurs in $O+A$ but not in O is a "new" statement arising from an insertion; when $O+A$ is the base program, we treat the missing s in O as if a user had deleted s from $O+A$ to create O. Version $O+A+B$ is still treated as being a program version derived from $O+A$. $O+B$ is created by integrating O and $O+A+B$ with respect to base program $O+A$.

We can illustrate this ability by pretending that after we created the program in buffer demoA (by editing a copy of the program in demoBase) we then created the program in INTEGRATED by editing a copy of demoA. We can now recreate the program in buffer demoB by integrating demoBase and INTEGRATED with respect to demoA. When we invoke the **integrate** command, we again get an empty parameter form, but this time we fill in demoA as the base program, demoBase and INTEGRATED as the two variants, and newB as the buffer into which the integrated program is to be placed.

```
Integration Form

Base program: demoA
Variant A:    demoBase
Variant B:    INTEGRATED
Integrated program: newB
```

We then invoke **start-command**. The integration of demoBase and INTEGRATED with respect to demoA succeeds, and the resulting program, placed in buffer newB, is identical to the one in demoB.

```
newB

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

Positioned at program
```

```
demoB

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

Positioned at program
```

4. ILLUSTRATING INTERFERENCE IN INTERFERING VERSIONS OF PROGRAMS

Some rudimentary diagnostic facilities for illustrating the conflicts that arise in an interfering integration have been incorporated in the prototype program-integration tool [6]. If interference is reported, it is possible for the user to examine sites of potential conflicts. (Roughly speaking, the sites reported are those at which slices of the two variants become "intertwined" in the underlying merged dependence graph.) The **slice** command can be invoked to provide further information about potential integration conflicts.

To illustrate these facilities, consider what happens if we try to integrate the programs in buffers demoA and demoB with respect to demoBase when the initialization statement i := 1 in demoA has been changed to i := 5, as shown below:

demoA

```
program demo;
begin
   sum := 0;
   i := 5;
   while (i <= 10) do
      sum := (sum + i);
      i := (i + 1)
   od;
   amean := (sum / (i – 1))
end.
```

Positioned at stmtList

With this modified version of demoA, the **integrate** command fails to produce an integrated program. (The buffer to which the integrated program has been directed receives the empty program.)

We can now examine sites of potential conflicts—sites that may or may not represent actual conflicts—using the commands **show-first-interference-vertex**, **show-next-interference-vertex**, and **show-previous-interference-vertex**. These commands move the selections of both demoA and demoB to common elements whose slices conflict (*interference vertices*). For example, **show-first-interference-vertex** causes the two selections to be moved to the while-predicate (i <= 10):

demoA

```
program demo;
begin
   sum := 0;
   i := 5;
   while (i <= 10) do
      sum := (sum + i);
      i := (i + 1)
   od;
   amean := (sum / (i – 1))
end.
```

Positioned at cond

demoB

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

Positioned at cond

Predicate (i <= 10) is an interference vertex because the predicate itself is computed in a different fashion in demoA than in demoBase, yet the slice of demoA with respect to (i <= 10) is not a compatible subslice of the slice of demoB with respect to statement gmean := (prod ** (1 / (i – 1))), which is a newly introduced statement of demoB. These two slices are shown below:

demoA

```
program demo;
begin
   sum := 0;
   i := 5;
   while (i <= 10) do
      sum := (sum + i);
      i := (i + 1)
   od;
   amean := (sum / (i – 1))
end.
```

Positioned at cond

demoB

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

Positioned at stmtList

There is also a second interference vertex that arises from the failed integration; command **show-next-interference-vertex** causes the two selections to be moved to the statement i := (i + 1):

demoA

```
program demo;
begin
   sum := 0;
   i := 5;
   while (i <= 10) do
      sum := (sum + i);
      i := (i + 1)
   od;
   amean := (sum / (i – 1))
end.
```

Positioned at stmtList

demoB

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

Positioned at stmtList

Statement i := (i + 1) is an interference vertex because the statement is computed in a different fashion in demoA than in demoBase, yet the slice of demoA with respect to i := (i + 1) is not a compatible subslice of the slice of demoB with respect to the (new) statement gmean := (prod ** (1 / (i – 1))):

demoA

```
program demo;
begin
   sum := 0;
   i := 5;
   while (i <= 10) do
      sum := (sum + i);
      i := (i + 1)
   od;
   amean := (sum / (i – 1))
end.
```

Positioned at stmtList

demoB

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

Positioned at stmtList

5. FINAL REMARKS

Among the obvious deficiencies of the present system are the absence of numerous programming constructs and data types found in full-fledged programming languages. Certainly one area for further work is to extend the integration method to handle additional programming language constructs. We have recently made progress towards handling languages with procedure calls [3] and pointer-valued variables [1]. Other language constructs, such as declarations, **break** statements, and I/O statements, as well as other data types will be addressed in the future.

It remains to be seen how often integrations of real changes to programs of substantial size can be automatically accommodated by the integration techniques developed in [2]. Due to fundamental limitations on determining information about programs via data-flow analysis and on testing equivalence of programs, both the procedure for identifying changed computations and the test for interference must be *safe* rather than *exact*. Consequently, the integration algorithm will report interference in some cases where no real conflict exists.

A successful integration tool will certainly have to provide facilities for programmers to cope with reported interference—facilities that would enable diagnosing spurious interference of the kind described above, as well as aids for resolving true conflicts. For these situations it is not enough merely to detect and report interference; one needs a tool for *semi-automatic, interactive integration* so that the user can guide the integration process to a successful completion.

Thus, future work on the prototype program-integration tool will aim to provide (1) better facilities for illustrating slices and the conflicts that arise in an interfering integration, and (2) additional capabilities for the user to resolve conflicts and create a satisfactory merged program. For example, renaming program variables and suppressing dependences between program components would be two ways a user might interact with the tool. Conflict-resolution facilities could also operate directly on the merged dependence graph, which is built by the integration algorithm whether or not the variants interfere [2].

ACKNOWLEDGEMENTS

I am indebted to Susan Horwitz for her comments as the system illustrated in the paper was developed. I would also like to thank Thomas Bricker for his role in implementing the system.

REFERENCES

1. Horwitz, S., Pfeiffer, P., and Reps, T., “Dependence analysis for pointer variables,” *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices* **24**(7) pp. 28-40 (July 1989).
2. Horwitz, S., Prins, J., and Reps, T., “Integrating non-interfering versions of programs,” *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).
3. Horwitz, S., Reps, T., and Binkley, D., “Interprocedural slicing using dependence graphs,” *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).
4. Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New York, NY (1988).
5. Reps, T. and Yang, W., “The semantics of program slicing,” TR-777, Computer Sciences Department, University of Wisconsin, Madison, WI (June 1988).
6. Reps, T. and Bricker, T., “Illustrating interference in interfering versions of programs,” *Proceedings of the Second International Workshop on Software Configuration Management*, (Princeton, NJ, Oct. 24-27, 1989), *ACM SIGSOFT Software Engineering Notes* **17**(7) pp. 46-55 (November 1989).