Conditions to Assure Semantically Consistent Software Merges in Linear Time

Keith Gallagher
Computer Science Department
Loyola College in Maryland
4501 North Charles Street
Baltimore, Maryland 21210
kbg@cs.louola.edu

Abstract

Software maintenance is the process of designing and integrating consistent changes to an existing software system. It is difficult for the maintainer to ascertain the complete effect of a code change; the maintainer may make a change to a program that is syntactically and semantically legal, but has ripples into the parts of the program that were to remain unchanged.

A decomposition slice is the union of certain program slices and can be used to decompose a program into a fixed part and and changing part. A decomposition slice induces guidelines for maintainers to use so that changes can be assured to be completely contained in the modules under consideration and that there are no undetected linkages between the modified and unmodified code. In this paper we review the conditions for changing on component in ways that will guarantee that the two components will not interact in any untoward fashion. What we have, then, are two related programs (since a slice is an executable projection). Adhering to these conditions permits the two related but semantically consistent programs to be merged in one pass over each source. In fact, the merge itself has no semantic component; it is equivalent to merging two sequences with repeated values eliminated.

1 Introduction

Decomposition slicing [3] provides the necessary framework to apply Weiser's technique of program slicing (usually considered only a debugging technique) to problems of software maintenance.

A decomposition slice, when viewed in the context of its original program, captures all the computation on a given variable and gives to the maintainer a list of the statements and variables in the projection that can be modified without impacting any code outside of the projection. Moreover, the program is *decomposed* into a changing part and a fixed part that is to remain unchanged by the maintenance activities. This fixed part, called the *complement* is also a decomposition slice. Since program slices are compilable and executable subsets of the program statements, we have two related yet different programs.

This paper gives conditions, induced by inspection of the relationship between the decomposition slice and its complement, that will permit the reconstruction of the modified program, in one pass over each source, in such a way that the complement remains fixed: the changes will be completely contained in one component of the decomposition. Thus a semantically consistent software merge can be done in linear time. Since maintainers are certainly free to add code, especial care must be taken to ensure that the fixed part does indeed remain fixed.

This paper is divided into 5 sections. Section 2 gives an overview of decomposition slicing and defines the requisite terminology. The results of section 2 have been presented elsewhere[4]. Section 3 gives the necessary conditions for a straight merge to be semantically consistent and describes the merging algorithm. Section 4 discusses error detection, that is, detection of violation of the necessary conditions. Section 5 places this particular effort in its appropriate setting by examining its larger context and discussing the relationship of this work to that of others in the same area.

2 Decomposing

2.1 Modifying Decomposition Slices

We now directly address the question: "What restrictions must be placed on modifications in a decomposition slice so that the *complement* remains intact?" We present the restrictions as a collection of rules that must be followed to answer the question.

Modifications take three forms: additions, deletions and changes. A change may be viewed as a deletion followed by an addition. We will use this second approach, and determine only those statements in a decomposition slice that can be deleted and the forms of statements that can be added. Again, we must rely on the fact that the union of decomposition slices is a slice, since the complementary criteria will usually involve more than one maximal variable.

Since independent statements do not effect data flow or control flow in the complement, we have:

Rule 1 *Independent statements may be deleted from a decomposition slice.*

This result applies to control flow statements and assignment statements. The statement may be deleted even

if it is an assignment statement that targets a dependent variable, or a control statement that references a dependent variable. The point to keep in mind is that if the statement is independent it does *not* effect the complement. If an independent statement is deleted, there will certainly be an effect in the slice. But the purpose of this methodology is to keep the complement intact.

There are a number of situations to consider when statements are to be added. We progress from simple to complex. Also note that for additions, new variables may be introduced, *as long as the variable name does not clash with any name in the complement.* In this instance the new variable is independent in the decomposition slice. In the following, *independent variable* means an independent variable or a new variable.

Rule 2 *Assignment statements that target independent variables may be added anywhere in a decomposition slice.*

This type of change is permissible even if the changed value flows into a dependent variable.

Adding control flow statements requires a little more care. This is required because control statements have two parts: the logical expression, that determines the flow of control, and the *actions* taken for each value of the expression. (We assume no side effects in the evaluation of logical expressions.) We discuss only the addition of **if-then-else** and **while** statements, since all other language constructs can be realized by them.

Rule 3 *Logical expressions (and output statements) may be added anywhere in a decomposition slice.*

We must guarantee that the statements that are controlled by newly added control flow do not interfere with the complement.

Rule 4 *New control statements that surround (i.e. control) any dependent statement will cause the complement to change.*

By making such a change, we have violated out principle that the complement remain fixed. Thus new control statements may not surround any dependent statement.

This short list is necessary and sufficient to keep the slice complement intact. This also has an impact on testing the change; again see [2] for details.

Changes may be required to computations involving a dependent variable, **v**, in the extracted slice. The maintainer can choose one of the following two approaches:

1. Use these techniques to extend the slice so that **v** is independent in the slice.

2. Add a new local variable (to the slice), copy the value to the new variable, and manipulate the new name only. Of course, the new name must not clash with any name in the complement. This technique may also be used if the slice has no independent statements, i.e., it is strongly dependent.

3 Merging the Modifications into the Complement

Merging the modified slice back into the complement is straightforward. A key to understanding the merge operation comes the the observation that the maintainer is editing the *entire program.* The Surgeon's Assistant gives a view of the program with the unneeded statements deleted and with the common statements restricted from modification. The slice gives smaller piece of code for the maintainer to focus on, while the rules of the previous section provide the means by which the deleted and restricted parts cannot be changed accidentally.

We now present the merge algorithm.

1. Order the statements in the original program. (In the following examples, we have one statement per line so that the ordering is merely the line numbering.) A program slice and its complement can now be identified with the subsequence of statement numbers from original program. We call the sequence numbering from the slice, the *slice sequence* and the numbering of the complement the *complement sequence.* We now view the editing process as the addition and deletion of the associated sequence numbers.

2. For deleted statements, delete the sequence number from the slice sequence. Observe that since only independent statements are deleted, this number is not in the complement sequence.

3. For statements inserted into the slice a new sequence number needs to be generated. Let P be the sequence number of the statement preceding the statement to be inserted. Let M be the least value in the slice sequence greater than P. Let $F = min(int(P+1), M)$. Insert the new statement at sequence number $(P+F)/2$. (Although this works in principle, in practice, more care needs to be taken in the generation of the insertion sequence numbers to avoid floating point errors after 10 inserts.)

4. The merged program is obtained by a straight merge of the modified slice sequence values (i.e. statements) into the complement sequence and eliminating repeated values.

Thus, the unchanged dependent statements are used to guide the reconstruction of the modified program. Again, this becomes clearer when the editing process is viewed

as modification to the entire program. The merge may be viewed as the insertion of the unmodified independent statements and newly added statements into the complement.

4 Error Detection

Error detection for the merging algorithm is done in two parts: before the merge starts and during the merge itself. The merger is provided with the following data: the original program, including a list of the declared variables; the decomposition slice, including independent and dependent statements, and and independent and dependent variables; and the the complement slice. The merger also assumes that no compilation errors have been introduced into the modified decomposition slice.

4.1 Errors Before the Merge

The Surgeon's Assistant [3] prohibits changes to dependent statements, but independent statements may be deleted without error. If such a facility were not available, a simple check that the dependent statements are intact would suffice. The line number generation feature guarantees that all new line numbers will be floats; in this way a dependent line could not be deleted and an unsuitable piece of code inserted in its place.

Variables newly introduced into the decomposition slice can be checked against the name space of the original program and the maintainer notified if such a case occurs. Note that if the modifier uses a new name that not is the decomposition slice, but is in the complement, a clean compile of the decomposition slice will occur.

Assignment statements may also be checked in this premerge pass to insure that only independent variables are targeted for assignment, and that no dependent variables are targeted for assignment *in new statements*. Output statements and logical expressions can be ignored.

4.2 Errors During the Merge

During the merge, we need to guarantee that no *new* control statements are themselves controlling any *dependent* statements. When processing newly added control statements, we perform a stack push. When the parse of the new control is finihed, the stack is popped. If any dependent statement is encountered while the stack is not empty, we have a violation: a control statement has a dependent statement in its controlled set of statements.

[Author's Note: We are aware that syntax-directed editing would obviate the need for an explicit merge, and that all errors herein described could be detected by a more robust modification tool. We're working on it!]

4.3 A Brief Analysis

The premerge activities can be mapped into the set membership operation. Using, say a b-tree representation of the set, the operations can be done in $O(log(n))$, where n is either the number of statements (for the independent statements check) or the number of variables in the program (for the name check and assignment statement checks). The the control statement check can be done while the merged program is being constructed and thus requires $O(n)$, where n is the total number of statements in the complement and the decomposition slice. Thus when the requirements are met we can perform a semantically valid merge in linear time.

5 Related Work

5.1 Testing the Change

Since the maintainer must restrict all changes to independent or newly created variables testing is reduced to testing the modified slice. This can be done since a program slice is an executable program. Thus the need for regression testing of values computed by the complement is eliminated. (See [2].)

There are two alternative approaches to verifying that only the change needs testing. The first is to slice on the original criteria plus any new variables minus any eliminated variables and compare its complement with the complement of the original: they should match exactly. The second approach is to preserve the criteria that produced the original complement. Slicing on this complementary criteria must produce the modified slice exactly as the complement of the complement.

5.1.1 Program Integration

Reps has embedded a program slicer and program integration facility in the Synthesizer Generator. This effort depends on an underlying representation of a program, called a *program dependence graph*. These graphs are characterized by having explicit representations of both data and control flow. Program dependence graphs are constructed for a restricted language that has scalar variables, assignment statements, conditional statements, and a special output statement that can only appear at the end of the program. The only variables that are defined at the end of the program are named in the output statement; any variable not referenced in this output statement is undefined at program termination.

Horwitz, Prins and Reps [5, 7] have constructed a program integration method for merging two variants of program with respect to the original. Whenever the changes made to the variants do not interfere, the result is a pro-

gram that integrates the two variants. Program slicing is used in the integration algorithm to determine possible interference (and non-interference). They have also applied this method for construct an environment for programming in the large; the techniques presented here would easily port to that paradigm. Reps and Bricker [6] have examined methods for illustrating the interferences.

5.1.2 Merging Software Extensions

Berzins [1] attacks the problem of combining independent updates to programs. His aim is to develop an automatic method for merging two versions of a program that guarantees the correctness of the merge or pinpoints potential inconsistencies. The semantics of the merged program are defined in terms of the semantics of the constituents. This work is limited to program extensions, but not program modifications. A program extension extends the domain of a partial function without altering any of the initially defined values, while a modification redefines values that were defined initially. That is, a program that results from merging two programs preserves the behavior of both; programs cannot be merged if they conflict at any points where both are defined. Berzins examines the conditions under which a textual merge corresponds to a semantic merge.

6 Conclusion

While some may view software maintenance as a less intellectually demanding activity than development, the central premise of our research is that software maintenance is *more* demanding. The added difficulty is due in large part to the *semantic* constraints that are placed on the maintainer. These semantic constraints can be loosely characterized as the attempt to avoid unexpected linkages. Berzins and Reps, et al. have addressed this problem by attempting to eliminate these semantic constraints and then providing the maintainer with a tool that will pinpoint potential inconsistencies after changes have been implemented. This makes maintenance appear to be more like development, since the programmer does not need to worry about linkages: once the change is made, the tool is invoked and the inconsistencies (if any) are located.

We take the opposite view: present the maintainer with a semantically constrained problem and let him to construct the solution that implements the change within these constraints. The semantic context with which we propose to constrain the maintainer is one that will *prohibit* linkages into the portions of the code that the maintainer does not want to change. This approach uncovers potential problems earlier than the aforementioned methods, and, we believe, is worth any inconvenience that may be encountered due to the imposition of the constraints. And, finally, as this paper shows, the solution to the problem of applying semantically consistent software updates in this constrained environment is equivalent to a straight merge.

References

[1] V. Berzins. On merging software extensions. *Acta Informatica*, 23:607–619, 1985.

[2] K. B. Gallagher. *Using Program Slicing in Software Maintenance*. PhD thesis, University of Maryland, Baltimore, Maryland, December 1989.

[3] K. B. Gallagher. Surgeon's assistant limits side effects. *IEEE Software*, May 1990.

[4] K. B. Gallagher and J. R. Lyle. Using program decomposition to guide modifications. In *Conference on Software Maintenance – 1988*, pages 265–268, October 1988.

[5] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. In *Proceedings of the SIGPLAN 88 Symposium on the Principles of Programming Languages*, January 1988.

[6] T. Reps and T. Bricker. Semantics-based program integration: Illustrating interference in interfering versions of programs. In *Proceedings of the Second International Workshop on Software Configuration Management*, pages 46–55, Princeton, New Jersey, October 1989.

[7] T. Reps and S. Horwitz. Semantics-based program integration. In *Proceedings of the Second European Symposium on Programming (ESOP '88)*, pages 133–145, Nancy, France, March 1988.