JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2003; **00**:1–7 *Prepared using* smrauth.cls *[Version: 2002/09/23 v1.02]*

Research

Guaranteed inconsistency avoidance during software evolution

Keith Gallagher[1] Mark Harman[2] and Sebastian Danicic[3]

[1] *Computer Science Department, Loyola College in Maryland, 4501 N. Charles St. Baltimore, MD. 21210, USA. kbg@cs.loyola.edu*
[2] *Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK. mark.harman@brunel.ac.uk*
[3] *Department of Mathematical and Computing Sciences, Goldsmiths College, University of London, New Cross, London SE14 6NW, UK. s.danicic@gold.ac.uk*

SUMMARY

The attempt to design and integrate consistent changes to an existing system is the essence of software maintenance. Software developers also confront similar problems: there are changes during testing, and the release of new system builds. Whether in development or maintenance, changes to evolving systems must be made consistently; that is, without damaging correct computations. It is difficult for the programmer to ascertain the complete effect of a code change; the programmer may make a change to a program that is syntactically and semantically legal, but which has ripples into the parts of the program that were intended to remain unchanged.

Using the standard denotational semantics for procedural programming languages, this paper formalizes decomposition slicing, which identifies interferences between software components and isolates the components to be changed. We enumerate the conditions for changing one component in ways that will guarantee that changes to it will not interact inconsistently and prove that changes made under these conditions are sound. Thus, the programmer can then execute changes secure in the knowledge that the semantics of the new system are guaranteed to be consistent with the projection of the semantics of the original for which it behaved correctly. Validating that the changes do not interfere not only guarantees consistency with respect to previous unchanging behaviors, but can also be achieved with complexity proportional to the size of the change to be made.

KEY WORDS: *Software evolution; program slicing; decomposition slicing; program transformation; inconsistency detection and avoidance*

*Correspondence to: K. Gallagher, Computer Science Department, Loyola College in Maryland, 4501 N. Charles St. Baltimore, MD. 21210, USA. kbg@cs.loyola.edu
Contract/grant sponsor: Commonwealth Scientific and Industrial Research Organization, Canberra, Australia
Contract/grant sponsor: Daimler-Chrysler
Contract/grant sponsor: EPSRC; contract/grant number: GR/R98938, GR/M58719, GR/M78083, GR/R43150

Received September 2002
Revised July 2003

INTRODUCTION

Software evolution is an activity that encompasses both development and maintenance. Evolution, change, is the essence of programming. In development, there is evolution. Even if one accepts the premise that new "development" starts with nothing (it rarely does), the construction of a new system is performed in a incremental manner. Subsystems and drivers are constructed; then consistently extended and integrated into the desired product. In fact, the modern process models for the construction of new software *insist* on an evolutionary approach; and, of course, software maintenance is about consistent change, consistent evolution.

The Problem and a Solution Method

The central difficulty of software evolution is the *semantic* constraints that are placed on the change engineer. These semantic constraints can be loosely characterized as the attempt to avoid unexpected linkages; i.e., making sure that effects of the change do not ripple into the parts of the code that were not supposed to change. We present the engineer with a semantically constrained problem in which to implement the change. The semantic context with which we propose to constrain the engineer is one that will *prohibit* linkages into the portions of the code that the engineer does not want to change. This approach uncovers problems early in the software change process, at compilation time. Without such an approach, the engineer must perform regression testing on the entire system (usually an impossible task), or use techniques that limit regression testing effort [1, 2, 3, 4].

Technical Summary

This paper *formalizes* previous work [5]; that is, it provides a sound denotational foundation to previously presented engineering principles. We create components, decomposition slices, by statement deletion from the original program. Each (executable) decomposition slice preserves a subset of the original semantics of the program, so we have a formal technique for restricting the partial function computed by the program to this subset. We give formal conditions between a pair of decomposition slices [6] that will permit the reconstruction of an evolved program in such a way that one of the decomposition slices remains unchanged. All changes will be completely contained in one component of the decomposition. The construction of the new program from the changed and unchanged parts can be done in time proportional to the size of the components.

The proof of the soundness of the conditions is framed in a control flow graph representation of the program; a parallel algorithm for static slicing [7] will be used in our approach, because it has the pleasant benefit of producing the equivalence class of *all* slicing criteria that are equivalent to the computed slice. That is, if a program slice on variable v at statement n yields statements $S = s_i; \ldots s_j$, the algorithm produces *all* slice criteria pairs (v_a, n_b) that have S as the slice. We exploit this attribute in the proof. We show that the time complexity

of the solution to the problem of applying semantically consistent software updates in this constrained environment is proportional to the size of the change.

Organization

This paper is divided into 5 sections of which this is the first. Section "Foundation and Context" formally defines the language used and summarizes the results upon which this work is based. This section reviews previous work, and is presented here for completeness. Section "Consistent Evolution," the major contribution of this work, gives the necessary conditions for a modification to be semantically consistent, proves their soundness, and discusses timing analysis. Section "Detailed Example" illustrates the techniques. The Conclusion discusses some implementation issues, assesses the impact of this presentation, and enumerates some ideas engendered by it.

FOUNDATION and CONTEXT

This section defines the language we use for the proofs. Subsections summarize previous work so that this presentation is self-contained and place this offering in its context.

THE LANGUAGE: Syntax

$$\begin{array}{lcl} I & ::= & Identifiers \\ E & ::= & Expressions \\ S & ::= & I := E \mid if\ E\ then\ S_1\ fi \mid \lambda \mid while\ E\ do\ S_1\ od \mid S_1; S_2 \\ \mathbf{P} & ::= & S; output \end{array} \tag{1}$$

The output statement also serves as a simple syntactic device to ease the representation of a slicing criterion which involves slicing at the end of the program. All slices will be taken at the output statement which will not be in the computed slice, per se, but will be displayed with the slice. λ denotes the empty statement. Evaluation of expressions has no side–effects. We permit the use of *for-statements* in our examples, using the usual syntactic transformation between *while*'s and *for*'s.

Defined and Referenced Variables

A state is a mapping from identifiers to their values, drawn from some set of denotable values, V. The set of all such states is denoted $\mathbf{S}$.

$$\mathbf{S} = I \rightarrow V \tag{2}$$

Let $\mathcal{M}$ denote the meaning of a statement. That is, $\mathcal{M}S$ is the state to state mapping denoted by S. M is a mapping from a program to its initial state, and from that initial state to the program's final state:

$$\mathcal{M} : \mathbf{P} \rightarrow \mathbf{S} \rightarrow \mathbf{S} \tag{3}$$

DEF and **REF** are functions from statements to sets of identifiers. $\mathbf{DEF}(S)$ is a safe approximation (a superset) of the identifiers which denote variables whose value changes for at least one statement in which S may be executed. $\mathbf{REF}(S)$ is a safe approximation (a superset) of the identifiers upon which the computation captured by S depends. That is, for a variable v in $\mathbf{REF}(S)$ at least one variable whose value is changed by S will depend upon the value of v.

$$v \in \mathbf{DEF}(S) \Leftrightarrow \exists \sigma \in \mathbf{S}.\mathcal{M}S\sigma v \neq \sigma v \tag{4}$$

$$v \in \mathbf{REF}(S) \Leftrightarrow \exists \sigma_1, \sigma_2.(\forall x \in I.x \neq v \Rightarrow \sigma_1 x = \sigma_2 x) \wedge \mathcal{M}S\sigma_1 \neq \mathcal{M}S\sigma_2 \tag{5}$$

The two definitions construct a syntactic (safe) approximation to defined and referenced variables, as captured above semantically. For example, the precise defined variable set for the statement `if TRUE then x:=1 else y:=1 fi;` contains only the variable $\{x\}$, while the approximation also contains `y`. Likewise, the precise referenced variable set for the statement `x := z - z` is empty, whereas the approximation contains z.

$$\begin{array}{lcl} \mathbf{DEF}(\lambda) & = & \{\} \\ \mathbf{DEF}(I := E) & = & \{I\} \\ \mathbf{DEF}(if\ E\ then\ S_1\ else\ S_2 fi) & = & \mathbf{DEF}(S_1) \cup \mathbf{DEF}(S_2) \\ \mathbf{DEF}(while\ E\ do\ S\ od) & = & \mathbf{DEF}(S) \\ \mathbf{DEF}(S_1\ ;\ S_2) & = & \mathbf{DEF}(S_1) \cup \mathbf{DEF}(S_2) \end{array} \tag{6}$$

REF can also be applied to expressions, where it yields the set of variables mentioned in the expression.

$$\begin{array}{lcl} \mathbf{REF}(\lambda) & = & \{\} \\ \mathbf{REF}(I := E) & = & \mathbf{REF}(E) \\ \mathbf{REF}(if\ E\ then\ S_1\ else\ S_2\ fi) & = & \mathbf{REF}(E) \cup \mathbf{REF}(S_1) \cup \mathbf{REF}(S_2) \\ \mathbf{REF}(S_1\ ;\ S_2) & = & \mathbf{REF}(S_1) \cup (\mathbf{REF}(S_2) - \mathbf{DEF}(S_1)) \\ \mathbf{REF}(while\ E\ do\ S\ od) & = & \mathbf{REF}(E) \cup \mathbf{REF}(S) \end{array} \tag{7}$$

Data can be arbitrarily complex and pointers are permitted. Handling composite structures and pointers requires careful definition of **DEF** and **REF**. For arrays, each assignment is both an assignment and a use of the array. For the statement `n: x[i] := v;` one may assume that $\mathbf{DEF}(n) = \{\mathtt{x}\}$ and $\mathbf{REF}(n) = \{\mathtt{i,\ v}\}$. However, the new value of `x` also depends on the old value of `x`, so `x` must also be included. The correct value for $\mathbf{REF}(n)$ is `{x,i,v}`. Records are simple and tedious. Unlike arrays, each field of a record is identified by a constant, the field's name. This allows occurrences of `record.field` to be treated as occurrences of the simple

variable `record_field`. The assignment of one record to another is modeled as a sequence of field assignments.

Multiple levels of indirection create difficult problems. One must obtain every possible location to which a pointer could point. If a variable is **DEF**ined or **REF**erenced through a chain of pointers (e.g., `*****a`), then all intermediate locations in the accessing of the variable must be in **REF**. Lyle, et al. [8] construct, then prune a *pointer state graph* for expression $*^k a$; see [8] for a detailed discussion and implementation.

The Parallel Slicing Algorithm

This section reviews [7] for completeness. The importance of this particular slicing algorithm for the present problem is that it not only constructs a slice, but records useful dependence information. This information is constructed as a by-product of the algorithm. It turns out that this information can be used to assess whether or not a change to a program will affect the semantic consistency.

The algorithm consists of three steps:

1. The original program is converted into a Control Flow Graph (CFG):
2. The CFG is compiled into a network of concurrent processes;
3. The network of concurrent processes is executed. The nodes of the CFG that correspond to statements to be kept in the final slice are obtained from the resulting output.

As the network is executed, dependence information is collected on the arcs of the CFG as sets of variable names. The presence of the set S on the arc A indicates that the value of every variable in S at A must be identical in every possible execution of the slice and of the original. It is this collection of variables that must remain the same (in a sense to be defined later) when a change is made.

The topology of the network is obtained from the reversed CFG (RCFG) – each edge corresponding to a channel of communication and each node of the RCFG corresponding to a process in the network.

Each process state consists of four–tuples. In the network diagram, the process state will be represented by placing these four state components in one of four quadrants. Defined variables will be placed in the lower left quadrant, referenced variables in the lower right, controlled nodes in the upper left quadrant and the processes identifier in the upper right. Figure 1 summarizes this and gives a sample node.

The process network constructed for program `variance.c` in Figure 2, is shown in Figure 3. Process communication is initiated by placing a message on the outgoing edge from the exit node; i.e., the entry node in the RCFG. In the network, all statements will have one input channel and at least one output channel. Node 8 is a predicate node, and thus has two output edges in the CFG. In the RCFG this corresponds to two input edges, thus the corresponding process has two input channels (and one output channel). Also, being a typical predicate, node 8 controls other nodes in the CFG, in this case nodes 9, 10, 11 and 12. Node 8 also references the two variables `ii` and `n` and has no defined variables.

The process behavior for an arbitrary process i is defined in CSP [9] in Figure 4. The formula is described as follows. To construct a slice for the criterion (v, n), network communication is

VALUE(S)	MEANING	POSITION
$C(i)$	The set of controlled nodes	Upper left
i	The identifier of corresponding node	Upper right
DEF(i)	The set of defined variables	Lower left
REF(i)	The set of referenced variables	Lower right

21 22 /* controlled nodes */	15 /* node id */
`var3` /* **DEF**(15) */	`x, ii, dev` /* **REF**(15) */

Figure 1. Contents of the nodes and sample layout.

initiated by outputting the set of messages in v from process n. For any process i, the node i should be included in the final slice if and only if process i has 'activated', that is, has output its node identifier i. This set of nodes can be found by outputting the set of nodes whose identifiers are input to the entry node, because the entry node is reachable via every node in the RCFG. Thus messages output by all nodes will eventually reach the entry process. No node identifier message can ever be removed from an edge once it has been output. The detailed proof of the termination of the algorithm is given in [7]. Informally, merely consider that no message placed on an arc can ever be removed; thus the labelings are monotone increasing functions with maxima.

The stable process state after variable `var5` is placed on the outgoing edge of the exit node is described in Figure 5. In the final state the arcs of the RCFG are labeled with identifiers and node numbers. The node numbers are those of reachable nodes which are in the slice; they are not useful in this context and not shown in the Figure. The variable names are useful, however. It is precisely the dependence information provided by this algorithm that provides that basis for the change techniques. For example, the presence of the name `t1` on the arc connecting nodes 4 and 5 indicates that the values of this variable must be preserved as execution passes from node 4 to 5. Therefore, no statement could be inserted which affects the value of this variable. However, a statement which assigned a value to some other variable could be inserted between these nodes. Similar arguments apply to the other variables on the edge.

When the network has stabilized, all arcs targeting the same node in the CFG must have the same labeling. The communication is initiated by a node in the RCFG firing its referenced variables. Thus, in the initial state of the network, all arcs entering the same node of the CFG are either empty or they target the slice node, in which case they are labeled identically by the referenced variables of the slice node. At each iteration, the only change to an arc labeling occurs when a node, n, fires its referenced variables, which it does on all output arcs in the RCFG. Therefore in the CFG, all input arcs to n are equally labeled, and they will remain so after the change in labeling.

```
          #include <stdio.h>
          #define MAX 1024
          main()
          {
                float x[MAX];
                float var1, var2, var3, var4, var5 ;
                float t1, t2, t3;
                float ssq, avg, dev;
                int  ii,jj, n;
/*  1 */           t3 = 0 ;
/*  2 */           t2 = 0 ;
/*  3 */           t1 = 0 ;
/*  4 */           ssq = 0 ;
/*  5 */           dev = 0;
/*  6 */           scanf ("%d", &n);
/*  7, 8, 12 */    for ( ii = 0 ; ii < n ; ii = ii + 1)
                   {
/*  9 */             scanf ("%f", &x[ii]);
/* 10 */             t1 = t1 + x[ii];
/* 11 */             ssq = ssq + x[ii] * x[ii];
                    }
/* 13 */           avg = t1 / n;
/* 14 */           var3 = (ssq  - n * avg * avg) / (n - 1);
/* 15 */           var4 = (ssq  - t1 * avg) / (n - 1);
/* 16 */           t1 = t1 * t1 / n;
/* 17 */           var2 = (ssq  -  t1 ) / (n - 1);
/* 18, 19, 23 */   for ( jj = 0 ; jj < n ; jj = jj + 1)
                   {
/* 20 */             dev = x[jj] - avg ;
/* 21 */             t2 = t2 + dev ;
/* 22 */             t3 = t3 + dev * dev ;
                   }
/* 24 */           var5 = (t3 - t2 * t2 / n ) / (n -1);
/* 25 */           var1 = t3 / (n - 1);

/* 26 */          printf("variance (two pass): %f \n",var1);
/* 27 */          printf("variance (one pass, using square of sum): %f \n",var2);
/* 28 */          printf("variance (one pass, using average): %f \n",var3);
/* 29 */          printf("variance (one pass, using average, sum): %f \n",var4);
/* 30 */          printf("variance (two pass, corrected): %f \n",var5);
          }
```

Figure 2. `variance.c` The statement numbering is referenced in Figure 3.

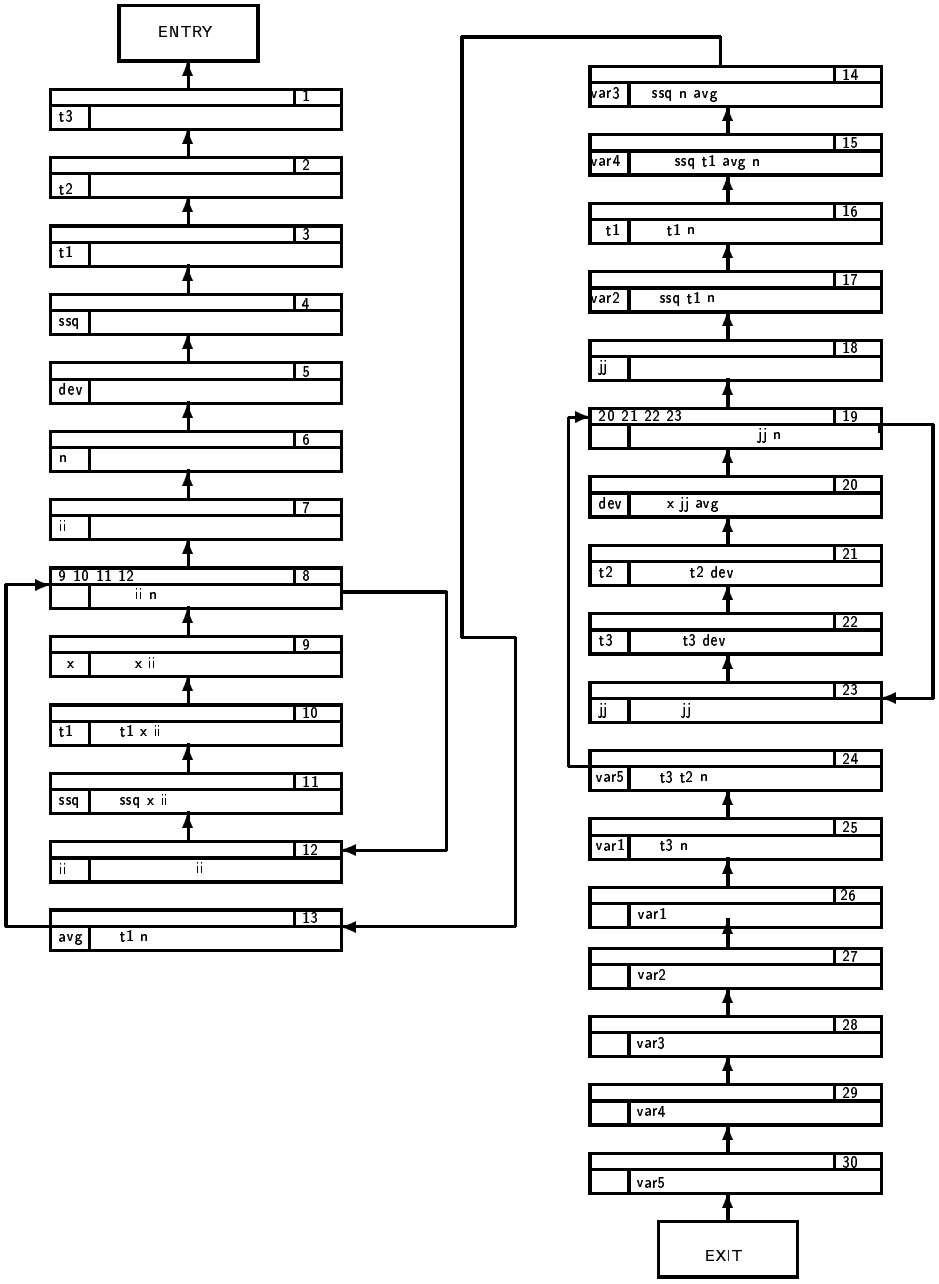

Figure 3. Initial Process State of `variance.c`.

$P(i) =$
$\quad ?\mathrm{S} \rightarrow$ (if $\mathrm{S} \cap (\mathbf{DEF}(i) \cup \mathrm{C}(i)) \neq \emptyset$
$\quad\quad$ then $!((\mathrm{S} \backslash \mathbf{DEF}(i)) \cup \mathbf{REF}(i) \cup \{i\})$
$\quad\quad$ else !S);
$\quad\quad P(i)$

Figure 4. CSP Description of Process Behavior

The arc labeling produced by this algorithm provides all the contextual information required about the program to be modified. The input to the entry node gives the slice. The labels between nodes enumerate the variables whose values must be preserved.

Decomposition Slicing

This sections reviews [6] for completeness. Decomposition slicing provides the necessary framework to apply program slicing to problems of software evolution. A program is *decomposed* into a changing part and an unchanging part that is to remain unchanged by the evolution activities. This unchanging part, called the *complement* is also a decomposition slice. A decomposition slice, when viewed in the context of original program from which it was derived, captures all the computation on a given variable and gives to the engineer a list of the statements and variables that can be modified without impacting any code in the complement.

A decomposition slice does not depend on statement numbers. The decomposition slice is the union of a collection of slices, which is still a program slice [10]. A decomposition slice captures all relevant computations involving a given variable and is defined as follows:

Definition 1 (Decomposition Slice)
Let

1. $Out(p, v)$ *be the set of statements in program* p *that output variable* v,
2. last *be the last statement of* p,
3. $N = Out(p, v) \cup \{\text{last}\}$.

The statements in $\mathcal{DS}(v, p) = \bigcup_{n \in N} \mathbf{SLICE}_{(v,n)}(p)$ *form the* decomposition slice on v.

In the language that we are using, output statements are permitted only at the last statement, so in this context $\mathcal{DS}(v, p) \equiv \mathbf{SLICE}_{(v,last)}(p)$. This is a considerable simplification. Decomposition slices were devised as an attempt to slice programs that had multiple outputs (on the same variable) at different locations in the program. A relatively straightforward series of program transformations can map a language that does permit arbitrary output statements (such as **C**) into our language. Requiring that output statements occur at the end of the program (and do not contribute to the value of a variable) precludes from our discussion programs in which the output values are reused, as is the case with random access files or output to files that are later reopened for input.

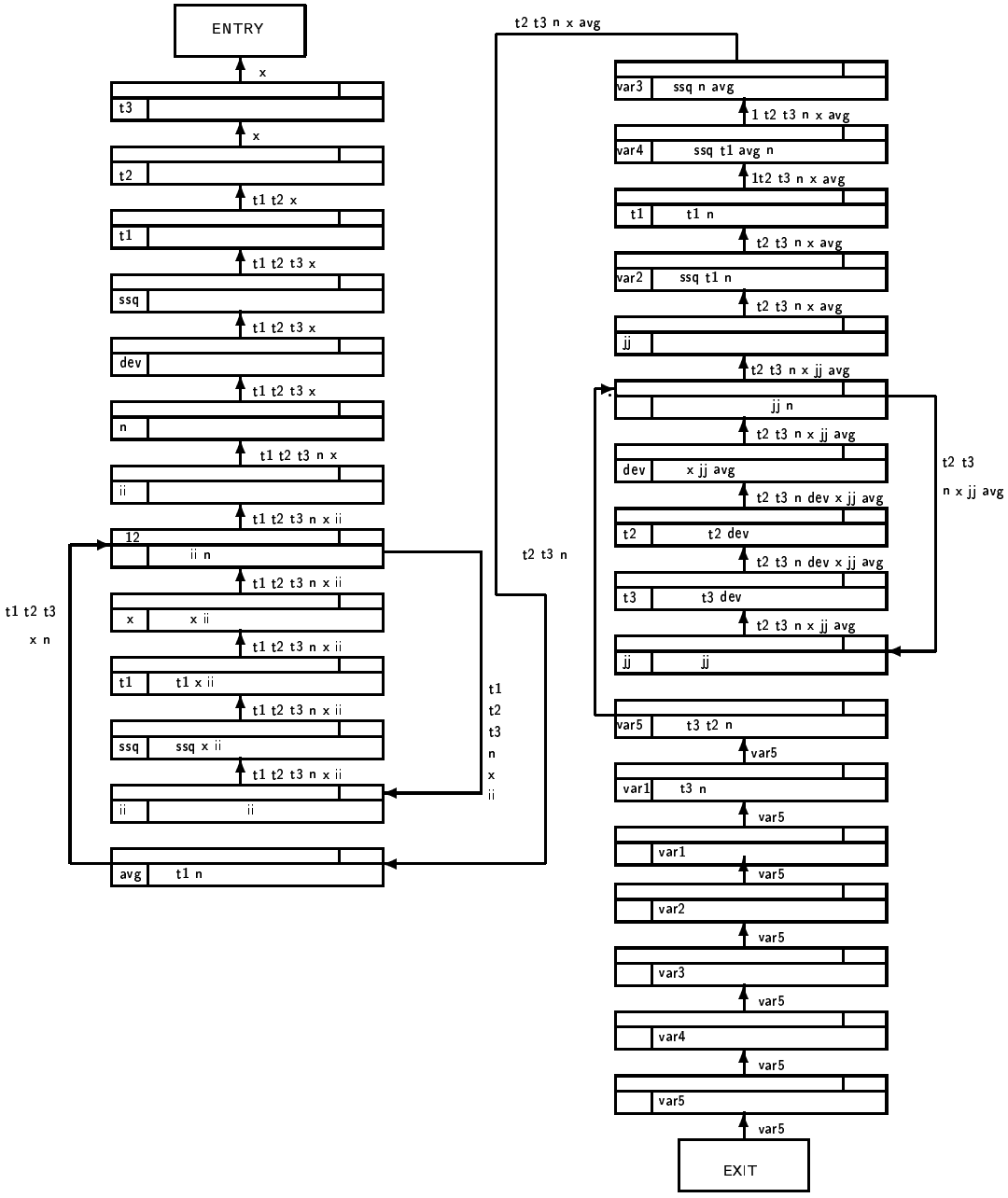

Figure 5. The stable process network of $\mathbf{SLICE}_{(var5,last)}(variance.c)$ i.e., $\mathcal{DS}(var5, variance.c)$.

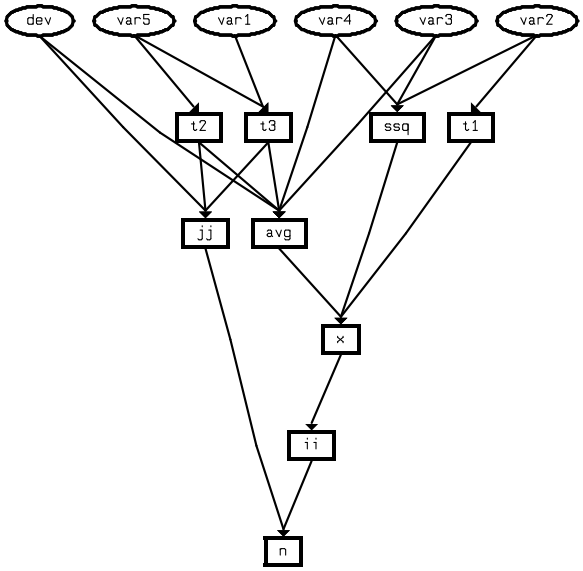

Figure 6. The graph ordering of decomposition slices with larger nodes at the top. The nodes show the variable names; the second parameter of $\mathcal{DS}(v, p)$, the program name, is not displayed.

We take the decomposition slice for each variable in the program and form a graph, using the partial ordering induced by set inclusion. (The term "lattice" was used in [6].) Larger sets will be toward the top; smaller sets will be lower; edges point downward. An edge between A and B means $B \subset A$ and there is no $C, B \subset C \subset A$.

Definition 2 (Maximal Decomposition Slices) *A decomposition slice $\mathcal{DS}(v, p)$ that is* **not** *contained in any other decomposition slice is said to be* maximal *(with respect to p).*

Maximal decomposition slices are at the top of the graph. Decomposition slices that are not maximal are said to be *dependent*. Figure 6 shows the graph of program `variance.c` of Figure 2. The decomposition slices $\mathcal{DS}(var1, variance.c)$, ..., $\mathcal{DS}(var5, variance.c)$, and $\mathcal{DS}(dev, variance.c)$ are maximal. The decomposition slices $\mathcal{DS}(avg, variance.c)$ and $\mathcal{DS}(ssq, variance.c)$, representing the average and the sum of the squares of the input, respectively, are dependent, as are all other interior points. It is interesting to note that the variable `dev`, representing the deviation of input value from the mean, was *not* output, even though it is maximal.

The *complement* of $\mathcal{DS}(v, p)$, $\mathcal{CS}(v, p)$, is formed by taking the union of all the maximal decomposition slices that are not contained in $\mathcal{DS}(v, p)$. This union remains a slice [10]. The complement is also a decomposition slice, so the terminology is complementary. For dependent decomposition slices, the complement is the entire program. When a decomposition slice and

its complement are examined, properties of the *statements* and *variables* are induced. We discuss these in turn.

Statements that are in $\mathcal{DS}(v,p) \cap \mathcal{CS}(v,p)$ (the intersection of a decomposition slice and its complement) are *unchangeable*. (The term "dependent" was used in [6].) The notation $\mathcal{UC}(v,p)$ is used to denote this set.

Both $\mathcal{DS}(v,p)$ and $\mathcal{CS}(v,p)$ can be written as sequence of unchangeable and changeable statements:

$$\mathcal{DS}(v,p) = D_0; \mathcal{UC}_1(v,p); D_1; \mathcal{UC}_2(v,p); D_2; \ldots; \mathcal{UC}_n(v,p); D_n$$

$$\mathcal{CS}(v,p) = C_0; \mathcal{UC}_1(v,p); C_1; \mathcal{UC}_2(v,p); C_2; \ldots; \mathcal{UC}_n(v,p); C_n$$

where $D_i \cap C_i = \emptyset$, for all i. Any of D_i or C_i may be empty. Statements in the set difference of the decomposition slice and its complement are *changeable*. (The term "independent" was used in [6].) Statements in this set difference are not even seen by $\mathcal{CS}(v,p)$; i.e,

$$\mathcal{CS}(v,p) \cap \mathcal{DS}(v,p) = \emptyset.$$

In particular, the D_i and C_i are changeable with respect to $\mathcal{DS}(v,p)$.

In this context, the problem of evolution may be considered from two complementary perspectives. An engineer may start with the idea of what needs to be changed, decompose and make changes to the decomposition slice so that the complement remains unchanged. Alternatively, the engineer may select what needs to stay the same, decompose, use this decomposition as the complement and make changes that hold this code invariant. In any case, the evolution problem may now be formally stated as "What restrictions must be placed on modifications in a decomposition slice so that the *complement* remains intact?"

Referring back to the program `variance.c`, the engineer may determine to change the computational method of `var2` (the top right of the lattice). This must be done without interfering with the computation of `ssq`; i.e., $\mathcal{UC}(var2, variance.c) = \mathcal{DS}(ssq, variance.c)$. It is also clear to see that the computations of `var3` and `var4` also depend on the computation of `ssq`. The engineer may determine to change the computation of `ssq`. In this case, the engineer knows the impact that this change will make: the new values of `var2`, `var3` and `var4` will need to be checked. However, the computations of `dev`, `var1` and `var5` do not need to be rechecked.

As a version of development, the engineer may start with the decomposition on `ssq` and proceed in one of two ways: add a new computation, `var6`, that uses `ssq`, but does not interfere with any other computations; or extract the decomposition on `ssq` (using the fact that slices are executable programs) to a new system that uses the sum of squares as a underlying computation. Note that if the extension is made in such a way that the decomposition on `ssq` does not change, we can easily integrate the different programs, as in [11, 12].

CONSISTENT EVOLUTION

We are now in a position to present the main result. First, we present a logical calculus of program transformations for program slicing. We then define rules in the program calculus

that use the arc labelings from the parallel slicing algorithm. The rules use the arc labelings to define consistent transformations with respect to the *complement*. Thus, previous engineering work has a basis in a formal language theory.

The transformation rules are written in the form of a logical calculus,

$$\frac{A}{\mathcal{T}[\![B]\!] = [\![C]\!]}$$

meaning that if A holds then it can be inferred that B can be consistently transformed into C. The transformation function $\mathcal{T}[\![]\!]$ takes a program and translates it into a syntactically altered, but semantically consistent program; i.e., the transformation so represented is consistent with the semantics of the *complement*.

When a program is modified, it goes through a syntactically invalid stage. This is due to the nature of typing. In the midst of typing the valid statement `a = b + c;` the program is not syntactically valid. We use the term *modification* to mean a complete, syntactically valid change.

Let $\mathbf{SLICE}_{(v,n)}$ denote the function which maps programs to slices with respect to (v, n). This function can be constructed using the algorithm of the previous section. The aim is to modify a program while preserving some slice. For the problem of consistent evolution, it will be the complement slice which is to remain invariant. Let $\mathcal{T}_{(v,n)}(p)$ denote the modified version of p which, whilst it may be different to p, defines the same slice when the slicing criterion is (v, n), thus $\mathcal{T}_{(v,n)}$ denotes the transformation function, which maps programs to programs while preserving the slice with respect to (v, n).

Definition 3 (Consistent Modification) *The modification embodied in* $\mathcal{T}_{(v,n)}$ *is* consistent with the slicing criterion (v, n) *iff* $\mathbf{SLICE}_{(v,n)} = \mathcal{T}_{(v,n)} \circ \mathbf{SLICE}_{(v,n)}$. *That is, slicing with respect to* (v, n) *is invariant under transformation according to* $\mathcal{T}_{(v,n)}$.

The Consistent Modification principle is about keeping the *complement* intact; the decomposition slice is to be changed. In order to satisfy this principle, it would be sufficient to simply slice the original program with respect to the desired slicing criterion and to re–slice the program each time it is modified. Any modification which alters the complement slice would be considered invalid and would be rejected. This solution, whilst attractively simple and demonstrably correct, is also highly inefficient, as it involves the recomputation of the complement for each and every modification. Modifications will be performed interactively, and are likely to be carried out little and often during evolution. Even if such modifications could be stored up, this would not be acceptable because modifications which do not preserve consistency will be disallowed. If the engineer attempts to make such a modification it will be important to catch this at the earliest opportunity so that an alternative, legal, modification may be selected. Thus, rather than re–slicing the program after a modification, it will be more convenient to determine, as cheaply as possible, whether or not a modification will affect a slice or not. The details of a simpler approach follow.

It will be convenient to associate the arc labels of the process network with nodes rather than arcs, so that arc labels can be depicted in textual representations of programs, an augmentation to a statement. The notation S^K for some primitive statement or predicate S (that is, a node) and set of variables K will be used to denote S, augmented by the arc labeling K,

where all outgoing edges in the RCFG (namely output arcs) are labelled K. In writing these transformation rules it is assumed that the program has already been sliced using the parallel algorithm to annotate statements with variable labelings and that the complement $\mathcal{CS}(v,p)$ of the program to be sliced is available.

Rule 1 (Sequential Insert) Sequential Insert allows a statement which cannot affect the complement to be added to the program. Observe that, if the insertion is legal then the new statement S, does not change the complement, and so, by definition, it cannot change the arc labeling, hence it is augmented with the arc labeling K.

$$\frac{\mathbf{DEF}(S) \cap K = \emptyset}{\mathcal{T}[\![S_1^K \ ; \ S_2^{K'}]\!] = [\![S_1^K \ ; \ S^K \ ; \ S_2^{K'}]\!]}$$

Sequential insert may be done in the middle of some $\mathcal{UC}_i(v,s)$. Suppose for some v and i, $\mathcal{UC}_i(v,s)$ = `a = b; b = c;` and $x \notin K_i$. Then the insertion if `x = b` between them, making two disjoint unchangeable regions, is permissible.

Rule 2 (Sequential Delete) A statement S may be deleted if it is not in the complement slice, $\mathcal{CS}(v,p)$.

Sequential Delete is simply the rule that captures the effect of determining changeable statements. Any statement not in the slice can be removed. In the case of sequential composition, these statements are identified by the fact that they do not define any variables on the arc which emerges from them in the CFG. This means that the arc labeling of such a statement will be identical to the arc labeling of that which follows it.

$$\frac{S \cap \mathcal{CS}(v,p) = \emptyset}{\mathcal{T}[\![S]\!] = [\![]\!]}$$

Sequential delete may collapse distinct $\mathcal{UC}_i(v,s)$ and $\mathcal{UC}_{i+1}(v,s)$ into a single sequence of unchangeable statements, by deleting the entire set of statements, C_i. However, rule 1 could be used to split them apart again.

Rule 3 (Control Capture) Control Capture occurs when a predicate-controlled computation is introduced which captures some portion of the original program. This is only legal if none of the captured portion occurs in the complement. Let $b(S)$ represent statement S controlled by predicate b.

$$\frac{\mathcal{CS}(v,p) \cap S = \emptyset}{\mathcal{T}[\![S]\!] = [\![b(S)]\!]}$$

This rule covers both *if b then* S_1 *else* S_2 *fi* and *while b do S od*.

Correctness of Rules for Consistent Evolution

In this section the rules for consistent evolution are proved to be sound. That is, if a change is made according to the rules, then this change must be consistent with respect to the

complement slice. The completeness result (that all changes which are consistent with respect to the complement slice are valid according to the rules) is not proved. Indeed the completeness result does not and cannot hold for any implementable set of rules, as this problem is not computable.

To see that the completeness problem is not computable, consider the simple program fragment below (in which `f` and `g` are arbitrary functions):

```
scanf("%d",&x) ;
if (f(x)!=g(x))  z=y;
```

Suppose the complement slicing criterion is {`z`}. Now, suppose that we wish to insert the statement `y=42;` at the start of the sequence. This will be allowable, if and only if the functions `f` and `g` are equivalent, as in this case the change will not affect the complement.

If a set of rules were defined which enabled us to decide whether or not such a change were allowable in all cases, then such a set of rules could be used to decide whether or not two arbitrary functions are equivalent. As the problem of deciding whether two arbitrary functions are equivalent is known to be undecidable, we must conclude that a complete set of computable rules could not be defined.

Fortunately, a sound (though incomplete) set of rules will still be safe because soundness guarantees that any change cannot affect the consistency of the complement. We therefore turn our attention to the problem of proving that our rules are sound.

Soundness of Consistent Evolution

The soundness proof is relatively straightforward. The parallel nature of the slicing algorithm means that we can construct the proof in terms of the changed behavior of a single process or sub-process network, since knowing the inputs to a process or sub-network means that we can compute its outputs. Furthermore, if we know the maximal input to a process function, then we can calculate its maximal output (as process functions are monotonic). Therefore, if we know the input to a process function in the stable network state, then we can calculate its output in the stable network state.

We shall consider each rule in turn.

Sequential Insert

The transformation defined by this rule allows us to insert a statement, S. Since the rule requires that $\mathbf{DEF}(S) \cap K = \emptyset$, we know that $(\mathbf{DEF}(S) \cup \mathrm{C}(S) \cap \mathit{Inp}) = \emptyset$ (where Inp is the input to the process which models S) and therefore, by definition of process behavior, we know that the output of the process which models S is identical to its input. Furthermore, we can see that S will not be included in the complement slice (by definition of process behavior).

Since the process which models the inserted statement S, copies its input to its output, the insertion of S can therefore have no effect on the behavior of the complement slice produced by the algorithm.

Sequential Delete

If a node, n is not included in a slice, then in the stable process network, the process which models n will copy its input to its output (this can be established by trivial induction on the CSP definition of process behavior). The rule for sequential delete requires that $S \cap \mathcal{CS}(v,p) = \emptyset$, which means that the statement to be deleted, S, is not part of the complement slice. The deletion of this statement therefore cannot affect the computation of the complement, as S is not in the complement and the process which models it copies input to output unaffected.

Control Capture

The rule requires that the statement, S, to be added to the program is not in the complement slice. Therefore the process which models S must copy its input to its output in the process network.

The transformation takes the sub-network consisting solely of the statement which models S to the subnetwork in which b controls the flow either around or through S. By definition, the input on each of the two input arcs to the predicate b in the RCFG will be identical to the input to S (because S must copy its input to its output).

As b is side–effect free it can only be included in the slice, if a node it controls outputs its node identifier. The only node which is controlled by b is, however, S and this is not included in the slice (and so will never output its node identifier). Therefore, the predicate b cannot be included in the slice and must pass the union of its input to its output. However, the union of the input to b is identical to the input to S and therefore the sub RCFG consisting of b and S has the same input–output behavior as the single node S and thus has no effect upon the slice constructed.

This concludes the soundness proof.

Timing

With the change S, arc labeling K and complement slice $\mathcal{CS}(v,p)$ in hand, the time to validate the preconditions of the rules is the time for calculating the intersection of K or $\mathcal{CS}(v,p)$ with S. This is linear in the order of the size of the modification to be made, S. Thus the modifications can be checked for consistency with negligible computational overheads.

DETAILED EXAMPLE

Background The task is to modify the **C** program `variance.c` of figure 2, which uses a number of techniques to compute the *variance* of set of values. Texts on statistics usually give a number of equivalent methods for calculating the estimated populations variance, s^2, based on a sample of n observations: ($\overline{x}$ is the average of the n data points)

$$s^2 = \sum_{i=1}^{n}(x_i - \overline{x})^2/(n-1) \tag{8}$$

$$s^2 = \left(\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2/n\right)/(n-1) \tag{9}$$

Formula 8 is the direct definition of variance, and requires two passes over the data set. In equation 9 the sum of squares $\sum_{i=1}^{n}(x_i - \overline{x})^2$ has been expanded and rearranged in a form that permits only one pass over the data set. The term $(x_i - \overline{x})$ is called the **deviation**. Equation 9 also has the following equivalent forms:

$$s^2 = \left(\sum_{i=1}^{n} x_i^2 - n(\overline{x}^2)\right)/(n-1) \tag{10}$$

$$s^2 = \left(\sum_{i=1}^{n} x_i^2 - (\overline{x}\sum_{i=1}^{n} x_i)\right)/(n-1) \tag{11}$$

The two pass method of equation 8 is recommended as the best for general use. When the standard deviation is small compared with the average, computational accuracy may be improved by making a correction to the sum of squares of deviations. The formula is

$$s^2 = \left(\sum_{i=1}^{n}(x_i - \overline{x})^2 - \left(\sum_{i=1}^{n}(x_i - \overline{x})\right)^2/n\right)/(n-1) \tag{12}$$

The correction using equation 12 is zero in pencil and paper computations; in automated computations, it estimates the error term. Note that the first term in the numerator of equation 12 is the *sum of* **squares** *of the deviations*, while the second term of the numerator of equation 12, the error estimate, involves the **square** *of the sum of the deviations.*

The Program The program, **variance.c**, in figure 2 computes the variance of a data set of maximum size 1024 using the five methods of the previous paragraphs. The data is read into array **x**, which is then used for the second pass. The first data value, variable **n**, is the number of data items which follow. No error checking is done. The outputs are as follows:

var1 gives the value using equation 8
var2 gives the value using equation 9
var3 gives the value using equation 10
var4 gives the value using equation 11
var5 gives the value using equation 12

That is, variable 1 uses the standard two-pass formulation; variables 2, 3, and 4 give the one-pass results; variable 5 uses the error estimate for correction. The variable names are mapped to the equation numbers.

The other variables are:

t1, t2, t3, ii, jj are accumulators and counters

ssq is the sum of squares

avg is the average, $\overline{x}$

dev holds the deviation of one data point from the mean.

The Change The changes are to add a new method for computing the variance and to use a different method to compute the error corrected two pass variance computation. All other computations are to remain unchanged. The new method uses an "on the fly" method for obtaining the average and the sum of the squares of the deviations. The formulæare given by the recurrence relations

$$m_0 = 0 \tag{13}$$

$$m_i = ((i-1)m_{i-1} + x_i)/i. \tag{14}$$

and

$$ss_0 = 0 \tag{15}$$

$$ss_i = ss_{i-1} + (i-1)(x_i - m_{i-1})^2/i. \tag{16}$$

This computes the *sum of* **squares** *of the deviations*. So, for equation 16, the variance is given by $ss_n/(n-1)$.

The error corrected two pass variance, equation 12, is to be replaced by equation 17 which uses m_n for both the error estimate and the sum of the squares of the deviations.

$$s^2 = \left(\sum_{i=1}^{n} (x_i - m_n)^2 - \left(\sum_{i=1}^{n} (x_i - m_n) \right)^2 / n \right) / (n-1) \tag{17}$$

Using the Rules

The stabilized process network, $\mathcal{DS}(var5, variance.c)$ and $\mathcal{CS}(var5, variance.c)$ are shown in figures 5, 7 and 8, respectively. The figure 7, $\mathcal{DS}(var5, variance.c)$, is annotated with the K sets *of the complement* as comments to the right of the respective statement. Note that the appearance of a $v \in K$ means that the listed values may *not* be changed prior to that statement. The changeable variables are `t2` and `var5`. The reader is invited to construct a solution under these constraints before continuing.

For the addition of computation of `var6`, three new variables `m`, `sqdev` and `var6` are introduced and assigned values using *sequential* insert: no new predicates are introduced. In terms of the formal definition of sequential insert, S is `m = 0; sqdev = 0;`. **DEF**(S) is $\{m, sqdev\}$. $K = \emptyset$ as the insertion of the initialization statements is made in front of the first statement. This is safe since neither of `m` or `sqdev` are in the annotated K set. The same rationale applies to the insertion of the accumulated terms inside the loop and the new computation of `var6`.

A tempting, but disallowed change is to modify `dev = x[jj] - avg` to `dev = x[jj] - m`, using the newly computed average, and then accumulate in `t2`. The engineer may not make this change since it violates both rule 2 and rule 1; it may be viewed as an attempted delete followed by insert. The attempted delete violates rule 2 since the statement `dev = x[jj] - avg` is in $\mathcal{UC}(var5, variance.c)$. This is also seen in comparing figures 7 and 8, $\mathcal{CS}(var5, variance.c)$ and $\mathcal{DS}(var5, variance.c)$. The attempted insert violates rule 1; the value of `dev` $\in K$ must be preserved for the following statement.

This is a microscopic illustration of one facet of the engineer's problem: part of the change is computed "far" from its use. When the new part of the change is subsequently used, it ripples into code where it was not intended, in this instance, the computation of `var1`. These rules require the engineer to construct a solution that does not alter `dev`. This lets the human be creative. If this error were introduced, it could only be discovered by a code review or a clever test. Both approaches certainly cost more than being automatically prevented from inserting the fault.

The result of merging the modified decomposition slice and the complement is in figure 9 and shows one possible solution. Two new variables `new_dev` and `new_t` are introduced and assigned values. The *choice* to use two variables was made by the engineer, who could have used only one. The variable `t2` is recomputed; i.e., deleted and inserted. The computation of `var5` is deleted by commenting, and a new formulation of `var5` is inserted. Note the similarity in the right hand sides of the graphs in figures 6 and 10. The decomposition slices on `var1`, `var4`, `var3` and `var2` are unchanged; $\mathcal{CS}(var5, variance.c)$ remains intact.

Comments

The solution can be criticized by two camps: one will maintain that in the addition of computation `var6` that intermediate variables should be introduced to hold the subcomputation of `x[ii] - m`; the other camp will criticize the change of `var5` for doing just that. We avoid these "religious" wars and offer an example of each style. Our argument is for human readability and understandability, issues that we recognize are fraught with religious overtones. However, we note that regardless of our implementation style, an optimized compiler emits object code that does not regard that style.

To compute `m` and `sqdev` it would also be correct to use another loop, as shown ing Figure 11. This is a safe method: one that guarantees that the newly added computations do not interfere with existing ones. This entire sequence of statements could be inserted using *sequential* insert. Another possible solution is based on the observation that the loop of Figure 11 has the same control as the first for-statement of lines 7–12. Thus, with a change of loop index, the accumulating assignments above could be placed in the first loop. The placement needs to

```
#include <stdio.h>
#define MAX 1024
main()
{
   float x[MAX];
   float var1, var2, var3, var4, var5 ;
   float t1, t2, t3;
   float ssq, avg, dev;
   int  ii,jj, n;
      t3 = 0 ;                              /* x */
      t2 = 0 ;                              /* t3 x */
      t1 = 0 ;                              /* t3 x */


      scanf ("%d", &n);                     /* t1  t3 x */
      for ( ii = 0 ; ii < n ; ii = ii + 1)  /* t1  t3 n x ii */
      {
        scanf ("%f", &x[ii]);               /* n ii */
        t1 = t1 + x[ii];                    /* n x ii */

       }
      avg = t1 / n;                         /* t1  t3 x n */



      for ( jj = 0 ; jj < n ; jj = jj + 1)  /* t3 n dev x jj avg var2 var3 var4 */
      {
        dev = x[jj] - avg ;                 /* t3 n x jj avg var2 var3 var4 */
        t2 = t2 + dev ;                     /* t3 n dev x jj avg var2 var3 var4 */
        t3 = t3 + dev * dev ;               /* t3 n dev x jj avg var2 var3 var4 */
      }
      var5 = (t3 - t2 * t2 / n ) / (n -1);  /* t3 n var2 var3 var4 */




     printf("variance (two pass, corrected): %f \n",var5);
}                                                 /* var1 var2 var3 var4 */
```

Figure 7. $\mathcal{DS}(var5, variance.c)$ with K sets *of the complement* annotated. The white space between statements is a visual indicator that statements have been sliced out.

```
#include <stdio.h>
#define MAX 1024
main()
{
      float x[MAX];
      float var1, var2, var3, var4, var5 ;
      float t1, t2, t3;
      float ssq, avg, dev;
      int  ii,jj, n;
         t3 = 0 ;

         t1 = 0 ;
         ssq = 0 ;
         dev = 0;
         scanf ("%d", &n);
         for ( ii = 0 ; ii < n ; ii = ii + 1)
         {
           scanf ("%f", &x[ii]);
           t1 = t1 + x[ii];
           ssq = ssq + x[ii] * x[ii];
          }
         avg = t1 / n;
         var3 = (ssq  - n * avg * avg) / (n - 1);
         var4 = (ssq  - t1 * avg) / (n - 1);
         t1 = t1 * t1 / n;
         var2 = (ssq  -  t1 ) / (n - 1);
         for ( jj = 0 ; jj < n ; jj = jj + 1)
         {
           dev = x[jj] - avg ;

           t3 = t3 + dev * dev ;
         }

         var1 = t3 / (n - 1);

        printf("variance (two pass): %f \n",var1);
        printf("variance (one pass, using square of sum): %f \n",var2);
        printf("variance (one pass, using average): %f \n",var3);
        printf("variance (one pass, using average, sum): %f \n",var4);
}
```

Figure 8. $\mathcal{CS}(var5, variance.c)$,

```
#include <stdio.h>
#define MAX 1024
main()
{
      float x[MAX];
      float var1, var2, var3, var4, var5;
      float var6, m, sqdev;
      float new_dev, new_t;
      float t1, t2, t3;
      float ssq, avg, dev;
      int  ii, jj, n;
      m = 0; sqdev = 0 ;

         t3 = 0 ;
         t2 = 0 ;
         t1 = 0 ;
         ssq = 0 ;
         dev = 0 ;
         scanf ("%d", &n);
         for ( ii = 0 ; ii < n ; ii = ii + 1)
         {
           scanf ("%f", &x[ii]);
           sqdev = sqdev + (ii*(x[ii]-m)*(x[ii]-m))/ (ii + 1);
           m = (ii*m + x[ii]) / (ii + 1);
           t1 = t1 + x[ii];
           ssq = ssq + x[ii] * x[ii];
          }
         var6 = sqdev / (n - 1);
         avg = t1 / n;
         var3 = (ssq  - n * avg * avg) / (n - 1);
         var4 = (ssq  - t1 * avg) / (n - 1);
         t1 = t1 * t1 / n;
         var2 = (ssq  -  t1 ) / (n - 1);
         for ( jj = 0 ; jj < n ; jj = jj + 1)
         {
           dev = x[jj] - avg ;
           new_dev = x[jj] - m;
           t2 = t2 + new_dev ;
           new_t = new_t + new_dev * new_dev ;
           t3 = t3 + dev * dev ;
         }
         var1 = t3 / (n - 1);
/*       var5 = (t3 - t2 * t2 / n ) / (n -1); */
         var5 = (new_t - t2 * t2 / n ) / (n -1);
        printf("variance (two pass): %f \n",var1);
        printf("variance (one pass, using square of sum): %f \n",var2);
        printf("variance (one pass, using average): %f \n",var3);
        printf("variance (one pass, using average, sum): %f \n",var4);
        printf("variance (two pass, corrected): %f \n",var5);
        printf("variance (one pass, square of deviation): %f \n",var6);
}
```

Figure 9. The changed program.

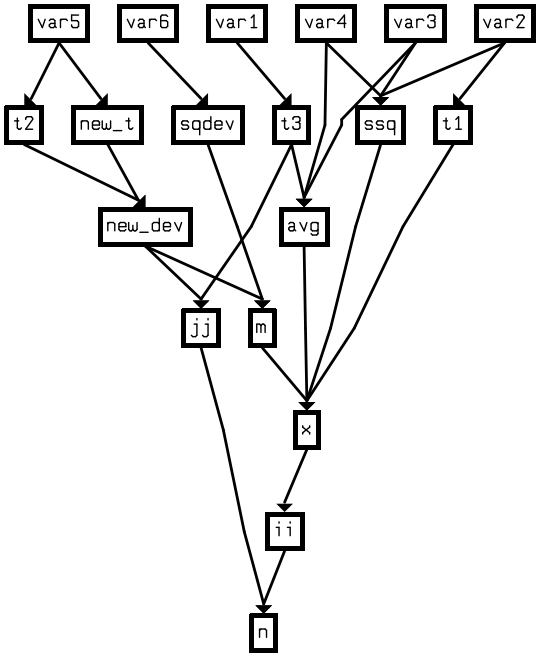

Figure 10. The graph of the changed program. Node `dev` has been hidden.

```
m = sqdev = 0;
for( kk = 0; kk < n ; kk = kk + 1)
{
  sqdev = sqdev + (kk*(x[kk]-m)*x[kk]-m)) / ( kk + 1 );
  m = (kk*m + x[kk]) / (kk + 1);
}
var6 = sqdev / (n - 1);
```

Figure 11. Complete sequential insert solution.

be after the `scanf`, but otherwise, as long as the above order is preserved, the placement is arbitrary. This, of course shortens the program by not adding another control structure. Since the proposed loop has the same abstract control structure as lines 18–23, it could also be merged into that loop.

The example does suffer from the fact the **Control capture**, rule 3, was not used. The one subject in a pilot study [13] who tried this method abandoned it. We admit that other solutions are possible. A solution which added another for-statement to pass over the stored data again, would be roundly criticized for doing just that, as a needlessly complex approach. However, such a solution, which uses rule 3, is correct. A clever compiler would optimize it into a form similar to the offered implementation. This could be presented as argument for another loop.

As an example of how casual errors may creep into a simple task, author KBG casually typed in the recurrences in the order presented in the document. This was wrong. The error was discovered by using the original program as an oracle to test the change, (the complement could have also been used as the oracle) and noting that the error could only be in one of the 3 *inserted* statements, and only these, because the changes used the rules. They were *guaranteed* not to interfere. The crucial point is that 3 consistent changes can be made, while inconsistent ones are avoided.

CONCLUSION

The Next Step

If we consider the analogy of surgery to software maintenance, this technique could be of of considerable value to a "software surgeon." It can be extended using by combining these principles with the technique of amorphous slicing [14, 15], that preserves semantic behavior, to perform "plastic surgery." Plastic surgery passes the program through a set of safe transformations that make it relatively easier to modify; inverting the transformations would then produce the new version.

For example, consider the example change. The loop of lines 7–12 of Figure 2 could be safely unrolled via safe transformations to 3 distinct ones, in which each value would be accumulated separately. Now suppose we add a *fourth* loop, using the same control structure, within which we compute the proposed change. Then we could re-roll the loops to get the changed program. As long as the transformations that unrolled and re-rolled the loops were safe, we have a way to further ease the semantic burden on the engineer.

Final Remarks

In essence, software maintenance consists of making changes to existing systems to correct or enhance their behavior. We support the argument that much of what is traditionally referred to as developmental activity is, in fact, a special case of maintenance. Notwithstanding this view, a large part of software engineering revolves around making changes to systems and ensuring

that these changes have few (preferably no) additional, unexpected or unwanted effects upon the system. We use the term "software evolution" to capture this combination of development and maintenance, though these results remain valid when a strict delineation is made between traditional developmental and maintenance activities.

In this paper we have used the decomposition slice and its complement to characterize the problem of consistent software evolution. Specifically, the complement of a decomposition slice abstracts from a system, a syntactic representation of the semantic component of the system which is to remain unchanged during the evolutionary life-cycle. In order to guarantee consistent evolution with respect to this complement, we introduce 3 rules which determine those changes that are consistency-preserving. This approach is linear with respect to the size of the change to be made. Our approach uses a slicing algorithm which computes not only decomposition and complement slices, but also additional variable dependency information. This suggests that an implementation of the rules as an "evolutionary consistency guarantor" will be relatively cheap to implement when compared with the rather high cost of regression testing. The software engineer who respects these rules will, of necessity, incur an inconvenience, because some kinds of change will be prohibited. (Clever engineers will be able to outsmart these prohibitions. The cost will be in additional regression testing in ensuring that there are no unintended linkages.) At this point the skill and experience of the engineer will become necessary to identify an alternative evolutionary path.

REFERENCES

1. Binkley D. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 1997; **23**(8):498–516.
2. Gupta R, Harrold MJ, Soffa ML. An approach to regression testing using slicing. *Conference on Software Maintenance 1992*. IEEE Computer Society Press: Los Alamitos CA, 1992; 299–308.
3. Rothermel G, Harrold MJ. A framework for evaluating regressions test selection. *Proceedings of the* 16^{th} *International Conference on Software Engineering*, ACM Press: New York, NY; 1994 201–210.
4. White L, Narayanswamy V. Test manager: A regression testing tool. *Conference on Software Maintenance 1993*, IEEE Computer Society Press: Los Alamitos CA, 1993; 338–347.
5. Gallagher KB. Conditions to assure semantically correct consistent software merges in linear time. *Proceedings of the Third International Workshop on Software Configuration Management*, ACM Press: New York NY; 1991 80–84.
6. Gallagher KB, Lyle JR. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering* 1991; **17**(8):751–761.
7. Danicic S, Harman M, Sivagurunathan Y. A parallel algorithm for static program slicing. *Information Processing Letters*1995; **56**(6):307–313.
8. Lyle JR, Wallace, Graham JR, Gallagher KB, Poole JE, Binkley DW. *A CASE tool to evaluate functional diversity in high integrity software*. U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg MD, 1995. http://hissa.ncsl.nist.gov/ [checked August 2003].
9. Hoare CAR. *Communicating Sequential Processes*. Prentice–Hall, 1985.
10. Weiser M. Program slicing. *IEEE Transactions on Software Engineering* 1984; **10**(7):352–357.
11. Binkley DW, Horwitz S , and Reps T. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology* 1995; **4**(1):3–35.
12. Horwitz S, Prins J,Reps T. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems* 1989 **11**(3):345–387.
13. Gallagher KB. Evaluating the surgeon's assistant: Results of a pilot study. *Conference on Software Maintenance 1992*, IEEE Computer Society Press: Los Alamitos CA, 1992; 236–244.

14. Harman M, Binkley DW, Danicic S. Amorphous program slicing. *Journal of Systems and Software* 2003; **68**(1):45–64.
15. Harman M, Danicic S. Amorphous program slicing. 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, IEEE Computer Society Press: Los Alamitos CA, 1997; 70–79.