

# Parallel Changes: Detecting Semantic Interferences

G. Lorenzo Thione  
FX Palo Alto Laboratory  
3400 Hillview Ave. Bldg. 4  
Palo Alto, CA 9304  
[thione@fxpal.com](mailto:thione@fxpal.com)

Dewayne E. Perry  
ECE & UT ARISE  
The University of Texas at Austin  
Austin, Texas 78712-1084  
[perry@ece.utexas.edu](mailto:perry@ece.utexas.edu)

## Abstract

Parallel changes are a basic fact of modern software development. Where previously we looked at *prima facie* interference, here we begin our investigation of a less direct form that we call semantic interference.

We reduce the forms of semantic interference that we are interested in to overlapping def-use pairs. Using program slicing and data flow analysis, we present algorithms for detecting semantic interference for both concurrent changes (allowed in optimistic version management systems) and sequential parallel changes (supported in pessimistic version management systems), and for changes that are both immediate and distant in time. We provide these algorithms for changes that are additions, showing that interference caused by deletions can be detected by considering the two sets of changes in reverse-time order.

## Introduction

As software development becomes increasingly geographically distributed and more work is done concurrently to meet market pressures, the need for tools to help in the process of parallel changes becomes more important.

In our previous study [1], we described the landscape of parallel changes, determined the extent of *prima facie* interfering changes, and established a positive linear relationship between the degree of parallelism and the number of software faults.

In this study, we present an analysis of parallel changes that have a less direct effect: *semantically interfering* changes. While all changes are intended to effect a program semantically, it is clearly the case that many changes have unintended effects – *cf* the fact that software faults are introduced in changes. [2,12] A change *semantically interferes* when modifications are made to the same slice [3,15] of the program and modify the program's behavior.

We delineate two different flavors of parallel changes in our approach to detecting semantic interference between any two versions.

- *Concurrent changes* are changes performed through an *optimistic version control* system which allow multiple valid copies of the same file at the same time [11]. The

final version is a semi-automated integrated version. The merging process is usually dependent on the absence of syntactic and direct *prima facie* conflicts; indirect semantic interference may easily go undetected [9]. Changes of this sort are truly parallel and there is no assumed ordering among them. Concurrent modifications pose several additional challenges in detecting interfering changes.

- *Sequential parallel changes* are also parallel changes in a logical sense: they take place independently from each other and are committed by different developers in a relatively short span of time. In a *pessimistic version management system* [7] parallel changes are sequentialized because only one developer may have a file checked out at a time. The time of submission for a change determines the ordering of the changes. By default, every change is assumed to possibly conflict with previous changes, potentially disrupting or modifying the effects of previous modifications, without notifying any of the developers.

In both cases, it is very likely that indirect semantic interference will go undetected. In related work, Horwitz et al. [9] provide an algorithm for integrating non-interfering versions under certain assumptions. To our knowledge, the extent to which this algorithm is applicable is unknown. Some version management systems [7,10,16] also provide automated merging functions. Nonetheless, as we discuss below, merging strategies find applicability only in optimistic version control scenarios and are nonetheless subject to various kinds of semantic interference.

## Semantic Interference

In analyzing semantic conflicts we focus on how variables are assigned and used throughout the data flow.

<pre>program a := 1  b := a end</pre>	<pre>program a := 1 a := 2 ← b := a end</pre>
---------------------------------------	---

Fig. 1 - A simplistic version of the *def-use static overlapping* case of semantic interference

The simplest example of semantic interference is the *def-use static overlapping* (Fig.1). The left hand snippet shows a simple definition-use pair. The only execution

path possible defines a value for the variable  $a$  and uses it in the next statement. As post-conditions on this code we expect  $a$  to carry the value 1 at the end of execution [9]. In the modified example an additional line has been inserted. The effect of this interference is clear: the variable  $b$  assumes the value 2 at the end of the execution, negating the post conditions that the author of the original code had intended. In this example we show only two versions of the code and refer to the leftmost as the *original code*.<sup>1</sup>

The next example shows a simple variation that uses aliasing to disguise the def-use overlapping. The variable  $a$ , a middle link in the dereferencing chain, is modified in value and although the original value for  $c$  is not modified,  $a^*$  and  $c$  do no longer refer to the same entity. Lines (2-4) and lines (3-4) constitute the overlapping def-use pairs.

	<b>program</b>		<b>program</b>
1:	$c := 1$		$c := 1$
2:	$a := \&c$		$a := \&c$
3:			$a := a + 1$ ←
4:	$b := a^*$		$b := a^*$
	end		end

Fig. 2 – A pointer variant.

While in the previous case the def-use pair involved assigning a value and accessing the location where the value was stored, in this case the interference occur during the pointer resolution process; in order to access the value of  $a^*$ , access to the value of the pointer is required. When line 3 is inserted, a new definition for the variable  $a$  is added. When the variable is used in order to resolve the value for  $a^*$ , interference occurs.

	<b>program</b>		<b>program</b>
1:	...		...
	$a := \&c$		$a := \&c$
2:	$a^* := 1$		$a^* := 1$
3:			$a^* := 2$ ←
4:	$b := a^*$		$b := a^*$
	end		end

Fig. 3 - Indirect effect.

Fig. 3 shows a simplified case of indirect semantic interference. A true indirect conflict occurs when despite the absence of apparent def-use overlapping involving the variables, the content of the memory location being referenced is modified before the value contained at that location can be accessed. While explicit pointers are the classic culprit of this type of interference, languages that implement implicit pointers (e.g. Java™, C#™ ) may simply obscure the conflict rather than preventing it.

In the specific example displayed in Fig. 3,  $a^*$  is an alias for  $c$ . Although there is no apparent def-use overlapping for any of the variables ( $a$  or  $c$ ), lines (2-4) and (3-4) still identify two overlapping def-use pairs for the entity referenced by  $a^*$ . Although this falls within the given definition for indirect conflicts, because the subject

entity is always referenced as  $a^*$  while in no case is the value of  $a$  affected, this example has more affinity with direct conflicts such as those shown in Fig.1 and Fig. 2. It is sufficient to consider  $a^*$  a stack variable, and static analysis of the code reveals the conflict.

Fig. 4 shows the simplest form of a truly indirect interference, for which no def-use overlap is detectable in the code without keeping track of pointers' values. The only variables present in both snippets are not explicitly affected by the change, although the value for the stack location indicated by  $c$ , to which  $a$  points, is changed through pointer dereferencing.

	<b>program</b>		<b>program</b>
1:	$c := 1$		$c := 1$
2:	$a := \&c$		$a := \&c$ ←
3:			$a^* := 2$ ←
4:	$b := c$		$b := c$
	end		end

Fig. 4 – A truly indirect semantic interference.

Indeed, there is no explicitly visible def-use overlapping for lines (1-4) and (3-4) which do interfere. For both versions, lines (1-4) apparently constitute a perfectly valid def-use pair and line 3, which involves a different variable, should not affect this. It is also evident though – at least in this simple example - that  $c$  is semantically an alias for  $a^*$  at this point in the execution, and that an overlap does indeed occur. This kind of semantic interference poses a series of additional challenges for detection purposes and is more problematic for developers to deal with.

The following variation shows the same interference through a different resolution scheme.

	<b>program</b>		<b>program</b>
1:	$c := 1$		$c := 1$
2:	$a := \&c$		$a := \&c$
3:	...		$c := 2$ ←
4:	$b := a^*$		$b := a^*$
	end		end

Despite no apparent use of the variable  $c$ , line 4 effectively accesses its value through a pointer, and line 3 introduces the semantic interference.

The ellipses in this example are intended to emphasize that while linking is apparent because of spatial proximity, the problem of determining what points to a certain variable is a difficult one which cannot be solved through purely static analysis of code. The ease of changing a variable while ignoring the in-links to that variable makes this kind of interference particularly critical.

We have so far discussed only changes that inserted lines which interfere semantically with a preexisting reference version. The same distinctions and classifications that apply to line insertions apply to changes that remove or modify any existing line of code. As we show below, the analysis of all modifications can be elegantly reduced to a combination or reversal of the

<sup>1</sup> We only identify two possible types of code modification: line insertions and line deletions; we will consider modifications as deletion-insertion pairs.

steps necessary for the detection of conflicts in the general case of line additions.

A semantic interference is therefore an artifact in the semantics of program evolution that defies the postconditions intended by the original author by interleaving pairs of def-use instruction in any of the program execution slices.

Although this generalization appears to hold in all situations, the actual interleaving mechanism may vary greatly. Pointer dereferencing can for instance involve multiple chained stages and interleaving def-use pairs may not appear as evident in these cases.

<pre> program 1: i,j,k:=0 2: a[0..2]:= {&amp;i,&amp;j,&amp;k} 3: 4: *a[i]=1 5: c:=j end </pre>	<pre> program i,j,k:=0 a[0..2]:= {&amp;i,&amp;j,&amp;k} *a[i]=1 ← *a[i]=1 c:=j end </pre>
--	---

Fig. 5 – A subtler example of def-use interleaving

The code of Fig. 5 shows that multiple def-use pair interleaving may occur when pointers dereferencing involves a chain of variables or aliases. Inserting line 3 introduces a definition of value for variable *i* before its use (line 4) but also changes the semantic of the subsequent assignment, causing line 4 to become a new definition for variable *j*.

While all flavors of semantic conflicts can be reduced to direct and indirect interference through def-use overlapping, the actual mechanisms in which these effects occur may also vary. In the examples shown so far, the execution path was unique and the interferences always occurred in the modified code. This is not always the case.

Control structures, subroutine calls, data-driven and event-driven execution schemes, all affect the number of different paths that the actual execution of code can take and thereby hide the effects of semantic interferences.

When non-determinism comes into play, as in multi-threaded operations, detection of undesired effects may be even more difficult.

In our experience with large version history data [1,12], a technique that reveals the locations of these side-effects would be extremely helpful in preventing the introduction of multiple new faults with every modification made to the code.

### Parallel Changes

Detecting semantic interferences in a real-world software development scenario entails first determining when two version changes are likely to interfere with each other unintentionally.<sup>2</sup> If every change were to be considered a possible source of semantic conflicts, then

every bug correction, and in general every modification request (MR) [1] would entail a semantically interfering change.

Generally speaking, we want to determine which changes are likely to interfere semantically although they were intended not to. Once the likelihood of unintentional interference is established, it is possible to ensure that such changes do not interfere with each other semantically, either directly or indirectly.

Changes are made to the code for several reasons. Mockus et al. [14] have identified and proposed several candidates for such reasons. Whichever its motivation, a change's explicit purpose is that of semantically affecting the previous version of the code, substituting it with a newer one. A change, though, may have side effects and semantic repercussions on other portions of the original code. Changes that affect the semantics of previous changes are interfering changes. While some changes interfere unintentionally because of side-effects unforeseen by all-too-human programmers, other changes (for instance corrective changes) purposely interfere semantically with some previously submitted change in order to correct a fault.

For our present purposes two changes are considered parallel and candidates for further inspection to determine whether or not they interfere when,

- for optimistic change management systems, when the changes submitted for integration are concurrent to each other. If the same module is undergoing simultaneous changes by several developers acting independently, changes performed to that module ought to be independent and not interfere semantically with each other.
- for pessimistic change management system, when the changes satisfy certain independency criteria, based on time of submission, spatial locality<sup>3</sup>, etc. For example, changes must be authored by different people, be tracked to different *initial modification requests* (IMRs), and not depend on either one's MR tree, etc.

In order to give an example of such definitions in the worst case of a pessimistic CMS, we consider the specifics of the 5ESS™ change history and version management system data. [1,12]

The 5ESS™ change process relies on an initial modification request (IMR) that models a logical problem, such as feature additions, fault corrections, performance improvements, etc. Each IMR is split into a set of more manageable modification requests (MRs),

<sup>2</sup> In concurrent changes this easily happens because the independent developers are unaware of each other's intentions.

<sup>3</sup> Semantic interference can propagate well beyond spatial locality, through function calls and library inclusion. On one hand, greater spatial distance in code strengthens independency claims, i.e. changes that are closer in space are more likely to be codependent. On the other hand, practical concerns suggest that reasonable boundaries be found. [17]

representing partial solutions to the problem, that reduce the granularity of the change being made. Each MR is typically owned by one developer, who implements it through a set of *deltas*. A *delta* specifies the change in terms of which lines of code were added, changed or deleted and includes pointers/references to the versions of code pre and post change [12, 2, 14].

In [19] we defined in greater detail a set of metrics used to determine how extensively two deltas purposely interfere semantically with each other. A function  $f$  is the probability function that estimates the probability that two deltas were meant to interfere.  $f=1$  will represent a (hypothetical) situation in which the two changes are certainly meant not to interfere semantically, while  $f=0$  will represent the opposite extreme, where the two changes are meant to interfere, i.e. a change correcting a fault introduced by another change.

We check two deltas for semantic interferences only when  $f(\delta_i, \delta_j)$  is greater than a chosen threshold  $\epsilon$ . The choice of using a threshold model for deciding when to apply the detection algorithm is suggested by strategic reasons in terms of cost, time and resources<sup>4</sup>. Also, such a threshold is a parameter whose primary effect is on the ratio between precision and recall<sup>5</sup>. The greater  $\epsilon$ , the less the changes to which the technique is applied, thus reducing recall and increasing precision. The smaller  $\epsilon$ , the more likely it is that conflicting changes are unintentional, reducing precision to the benefit of a larger recall.

Once we have determined two deltas that may unintentionally interfere semantically, we examine the different versions produced by these deltas when submitted to the change management system.

In our reference system, single MRs are split into a multitude of atomic changes (the deltas). One could argue that a MR produces one functionally consistent version of the software code<sup>6</sup>, and that we should focus on versions produced by MRs rather than by single deltas. Single deltas traceable to different MRs interleave changes to the code throughout the system's evolution in the sequential case. Therefore we consider *deltas* as the atomic originators of semantic interferences in parallel changes rather than MRs. This fine level of granularity however does introduce several complications addressed below.

Fig. 6 shows how changes from different MRs create versions of the code that overlap across MRs throughout time<sup>7</sup>. In general an arbitrary number of concurrently

active MRs generate deltas applied in a non-foreseeable order. The study of two generic deltas for deciding whether the later interferes semantically with the earlier, must involve the analysis of four different versions of the source code about which no particular assumptions can be made. Only when two deltas are consecutive will the intermediate versions be identical. In general no other change occurred between the deltas under observation.

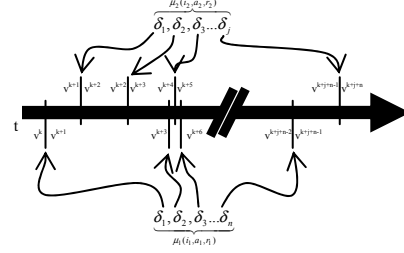


Fig. 6 – Deltas from different (independent) MRs, interleave non-deterministically in time. Adjacent deltas in a MR cannot be guaranteed to operate on adjacent versions of the code.

There is only one exception to this. In the case of truly concurrent changes in optimistic CMSs deltas are checked before integration (or contextually to it). There is therefore no possibility that other deltas occur between those under analysis. While this reduces the number of versions to three, the merged version must be brought into the process in order to successfully “catch” any conflicting behavior.

### Interference Detection: Conflicting Versions

The next and computationally most intensive step of our analysis technique involves *slicing* the code associated with each of the four versions required by a pair of deltas.

A *program slice* [3,4] represents the source code components of a software program that could potentially affect the semantics of the computation depending on a *slicing criterion*, usually a 2-tuple (program point, set of variables) and the task of computing program slices is called *program slicing*.

An important distinction is to be made between *static slicing* and *dynamic slicing*. Whereas static slices are computed without making assumptions regarding a program's input, and thereby include all potentially affected program points, dynamic slicing relies on specific test cases, on a specific set of input values for the program [3]. A program point is either an explicit instruction in the code such as an assignment, a control or a function call, a control point such as the entry point for a subprocedure, or a “hidden” program point such as assignment for formal/actual parameters and return values from subprocedures [4].

A Program Dependency Graph (PDG) is a directed graph for a single procedure in a program. Several PDGs representing the control components (subprocedures, etc.)

<sup>4</sup> The algorithm we describe is based on computationally intensive steps, such as code slicing and computation of System Dependency Graphs

<sup>5</sup> Precision corresponds to the fraction of semantic conflicts correctly detected over the overall number of pairs to which the process is applied. Recall reflects the fraction of deltas to which the process is applied over the total number of parallel changes. Tweaking the threshold is crucial to manipulating precision and recall.

<sup>6</sup> This is because one MR, owned by one developer, groups changes that have one common reason and stem from one initial logical request of modification (IMR)

<sup>7</sup> Note that the diagram assumes only two MRs for which interleaving deltas are submitted.

of a system are interconnected in a System Dependency Graph (SDG). The vertices of such graphs are program points while the edges represent control and data dependencies. Computing a program slice is the process of determining a subset of the vertices of one SDG whose members influence, or are influenced by, a particular slicing criterion. This duality corresponds to the choice of computing a forward or a backward slice. [3] The intersection between a forward and a backward program slice is called a chop between two program points. [4]

Fig. 7 shows a flow-chart of the steps involved in detecting semantic conflicts among parallel changes. The process assumes that two deltas have been chosen; the data available are therefore two chosen deltas,  $\delta_1$  and  $\delta_2$ .

Here we will describe the process step by step, through the aid of examples. For simplicity, we will start by discussing the general case in a pessimistic change management system. The general case also applies when two non-concurrent deltas are deemed parallel in an optimistic CMS environment.

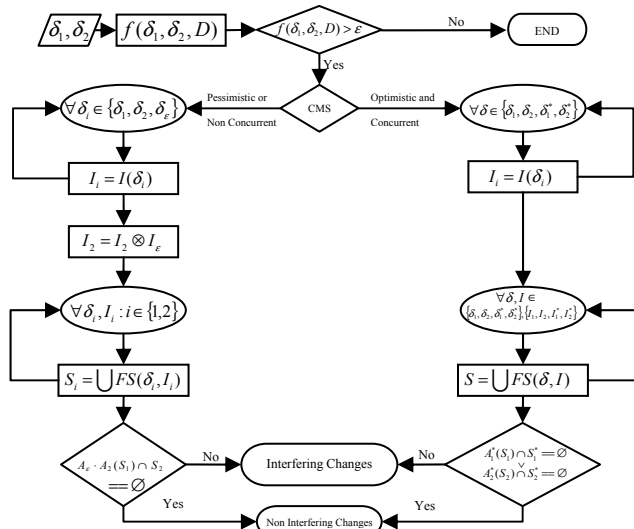


Fig. 7 – The flow diagram for the interference detection process<sup>8</sup>.

As discussed earlier, for two generic deltas,  $\delta_1$  and  $\delta_2$ , ordered in time<sup>9</sup>, four different versions of the software system under development are referenced, ordered progressively according to their time sequence. If two deltas are greatly separated in time,  $V_2$  and  $V_3$  (i.e. the version generated by the older delta and the one from which the newer in time starts) will largely be different. Only when the two deltas are adjacent will these versions be identical (Fig. 6). This distinction will be of particular relevance further on in the analysis.

Next, we construct a fictitious delta,  $\delta_e$ , which describes the sum of the changes made in the time lapsed between the two deltas under analysis. It can be simply shown that the combination of deltas is a closed operation in such space, because the subsequent application of deltas can be seen as a combination of line insertions and line deletions. The role of  $\delta_e$  will be key in determining possible interferences between the two deltas.

Reviewing the different types of semantic interferences and showing how they all could be related to the def-use insertion, we provided a clear understanding of what two interfering versions of code were. The central question that then arises is when do two deltas, rather than two versions, interfere semantically?

We now raise the question of when precisely two deltas can be said to be interfering. First, the interfering relation is an ordered one. We are interested in verifying – at least in a non-concurrent situation – that a delta applied at a later point does not interfere with a previously submitted delta assumed to be parallel to it (in the sense earlier defined earlier).

To do so we must establish the extent to which the effects of a change were intended by the author, while identifying any other side effect. We return to this point shortly. First, however, we will seek to define more formally how versions interfere semantically with each other.

Let  $V$  be a version of the source code of a software program. We can define the dependency set  $\Delta$  as the set of the triples  $(v, d, u)$  as follows,

$$\Delta = \{v : d \mapsto u\}$$

where  $v$  is a variable,  $u$  is a use of such a variable and  $d$  is the definition of the value for  $v$  that is effectively used in  $u$ . Such a set is fairly straightforwardly derivable from the program's SDG.  $d$  and  $u$  are by all definitions program points in the software dependency graph. In the simple examples used within this scope, they will always translate to source lines and will therefore be referenced by means of line numbers. Bear in mind however that several program points are often generated from the same source. Therefore program points in an SDG may show a one-to-one correspondence with source code lines.

For each delta  $\delta$  then, two dependency sets  $\Delta_1$  and  $\Delta_2$  can be determined, as well as an injection  $A : SDG_1 \rightarrow SDG_2$  that transforms each program point in  $V_1$  into a program point in  $V_2$ , thus codifying the changes in terms of program points.

Now, it should appear clear that the application  $A$  can only be an injection in  $SDG_2$  when the only changes made from  $V_1$  to  $V_2$  are line insertions. Since a line change is a deletion followed by an insertion, it remains to be defined what happens in case of removed lines. A line deletion from  $V_1$  to  $V_2$  with  $t_2 > t_1$  is equivalent to a line insertion where  $V_1$  follows  $V_2$  in time. For the

<sup>8</sup> The process shown is applied to two given deltas which are analyzed for interference. Note that two deltas that are concurrent in an optimistic CMS scenario always qualify as parallel changes, but in general two non concurrent changes may qualify as parallel and analyzed following the left branch of the diagram.

<sup>9</sup> Assuming  $t_{\delta_1} < t_{\delta_2}$

purpose of determining semantic interferences a deletion is equivalent to an insertion reversed in time. In order to determine whether or not  $V_2$  interferes semantically with  $V_1$  over a set of deletions, we must therefore establish if  $V_1$  interferes semantically with  $V_2$  over the equivalent set of line insertions.

Consequently, the application  $A$  is logically an injection from  $SDG_1$  to  $SDG_2$  where the latter is the potentially interfering graph and the former is the one that is potentially interfered with.

In determining a condition of non-interference for versions  $a$  and  $b$  in Fig. 8, we argue that *a version interferes semantically with another when for any preserved use of a common variable, the immediate data predecessor or definition thereof, is not preserved.*

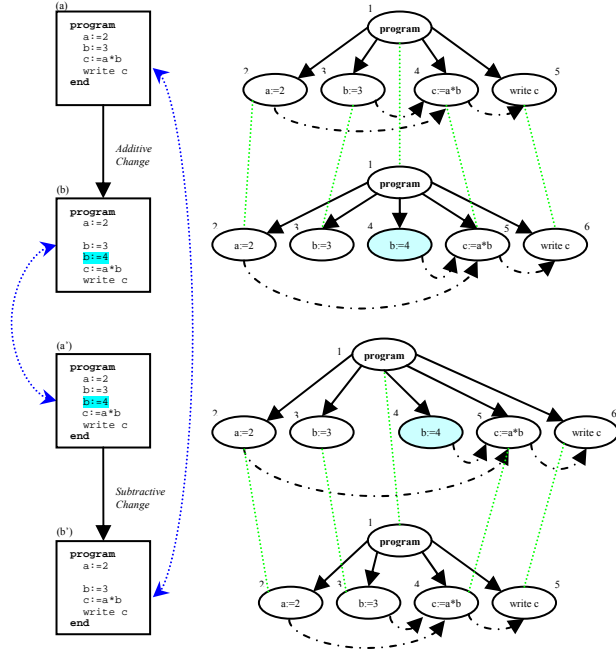


Fig. 8 – A simple program in two interfering versions and the corresponding SDG shown for additive and subtractive types of changes. The solid connectors represent flow dependencies, the black dashed ones represent data dependencies and the green pointed connectors show the application  $A$  from graph to graph. The blue connectors show the equivalence between the programs by looking at subtractive changes as additive changes in a reversed timeframe. Note that  $A$  is an injection in the case of additive changes (every node in (a) is connected to some node in (b)) and the inverse injection in case of subtractive changes (some node in (a') are connected to nodes in (b') exhausting the node space in (b')).

In this example, the variable  $b$  is common to both versions and a use of such variable is preserved in the program point pair (4,5) linked through application  $A$ . The immediate data predecessor relative to  $b$  - (3) for the former version and (4) for the latter - are not linked via  $A$  and therefore the variable definitions concerning a preserved use do not apply.

The latter version, however, does therefore interfere semantically with the former. Let  $V_2$  and  $V_1$  be two versions of code. Consider the case in which the modifications made to the code are generic, consisting of insertions, deletions and modifications. We must therefore verify that  $V_1$  does not interfere with  $V_2$  over the

set of line deletions, and that  $V_2$  does not interfere with  $V_1$  over the set of line insertions. We do so using the following condition of non interference.

Let  $V = V_1 \cap V_2$  be the intersection of the sets of variables used in  $V_1$  and  $V_2$ , and  $A_{V_1, V_2}$  be the injection from  $V_1$  to  $V_2$ .<sup>10</sup>  $\forall v \in V, \forall u_1, \exists (u_1, d_1, d_2), (u_1, u_2) \in A_{V_1, V_2} :$

$$[v : d_1 \mapsto u_1] \in \Delta_1 \wedge [v : d_2 \mapsto u_2] \in \Delta_2 \Rightarrow (d_1, d_2) \in A_{V_1, V_2} \quad (NI_V)$$

The condition  $NI_V$  is thus: *for every variable  $v$  common to both versions and for each use of that variable made in the latter version, if there exist an  $A$ -equal use in the dependency set of the former version, governed by an  $A$ -equal definition, the latter version does not interfere with the former.*

It could seem that the distinction between additive and subtractive changes is unnecessary and that the above definition could be applied regardless of the nature of the changes being made. While this may sometimes be the case, it does not hold when the cardinality of the dependency sets is not maintained<sup>11</sup>. This may happen when programs show more complicated flow diagrams than those we have shown. The versions of the program in Fig. 9 interfere semantically. The latter version removes two lines from the original version. Nonetheless, were  $NI_V$  to be directly applied, no interference would be caught. Since the changes are subtractive, in order to be sure to catch all possible interferences,  $NI_V$  has to be applied switching the versions.

Once again, for sake of clarity and simplicity we refer to program points in the following example by means of line numbers, binding each program point to the source code line that generated it.

1:	program(b)		program(b)
2:	a:=0		a:=0
3:	c:=0		c:=0
4:	if (b==0)		if (b==0)
5:	a:=2		a:=2
6:	else	←	fi
7:	c:=2	←	d:=a*b*c
8:	fi		write c
9:	d:=a*b*c		end
10:	write d		
	end		

Fig. 9 – A change that removes lines from a precedent version ought to be checked for interference by applying the non-interference condition  $NI_V$  switching the role of each version. This is equivalent to check for interference in line additions by considering a reversed timeframe.

A relatively straightforward analysis of the SDG of each of the two versions will reveal that:

$$\Delta_1 = \{[a : 2 \mapsto 9], [a : 5 \mapsto 9], [b : 1 \mapsto 4], [b : 1 \mapsto 9], [c : 3 \mapsto 9], [c : 7 \mapsto 9], [d : 9 \mapsto 10]\}$$

$$\Delta_2 = \{[a : 2 \mapsto 7], [a : 5 \mapsto 7], [b : 1 \mapsto 4], [b : 1 \mapsto 7], [c : 3 \mapsto 7], [d : 7 \mapsto 8]\}$$

Since the dependency set is computed statically, one variable can allow multiple definitions to one use. In our approach code is sliced statically. There is therefore no way to predict what the data that drives the execution

<sup>10</sup> Note the asymmetry between the quantifiers scoped over  $u_1$  and  $u_2$ . This asymmetry is what drives the rationale behind the distinction between additive and subtractive changes.

<sup>11</sup> i.e. dependencies are not simply modified but are introduced or removed

flow will be. In fact, if we sliced the code dynamically, the cardinality of each dependency set would be exactly equal to the number of uses made of all variables involved in that execution path, and therefore for every preserved use only one dependency in each set could be found.

An analysis of the information relative to the changes made (available through the change management system and associated with each delta) defines the function  $A$  as:

$$A = \{(1,1), (2,2), (3,3), (4,4), (5,5), (6,\varepsilon), (7,\varepsilon), (8,6), (9,7), (10,8)\}$$

To try to determine whether  $V_2$  interferes with  $V_1$  by applying  $NI_v$  directly would erroneously lead to the conclusion that the two versions do not interfere. In fact for each preserved use<sup>12</sup> of  $c$  in  $V_2$ , the relative definitions (3,3) are preserved. However, if  $b$  equaled 0 in  $V_1$   $c$  would carry the value 2 instead of 0.

Simply reversing the roles of the dependency sets will do the trick. The universal quantifier is scoped over the version from which the lines were removed guaranteeing that the interference will be discovered. Since this phenomenon is triggered by the subtractive change, we argue that  $NI_v$  must be applied in a reverse fashion for line deletions. In fact we argue that a deletion is a line insertion in a reversed timeframe. If a version subject to subtractive changes would interfere with modifying version when the time flow is considered inversely, then the deletions in the actual latter version causes semantic interferences by virtue of those deletions.

### Interference Sets: Conflicting Deltas

We have at this point defined a formal condition for non interfering versions. We now define an **interference set** which describes how two versions interfere with each other. Such set represents the interferences that the latter version in time induced over the former independent of how  $NI_v$  is applied.

For a generic set of line insertions, deletions and modifications between two versions  $V_1$  and  $V_2$ , one set of deletions and one set of insertions can be constructed. We will call these two sets  $C_{Del}$  and  $C_{Ins}$ .

We start applying  $NI_v$  to  $(V_{Del}, V_1)$  where  $V_{Del}$  is the version of code generated from the changes from  $C_{Del}$  applied to  $V_1$ , i.e.  $C_{Del}(V_1)$ . We then apply  $NI_v$  to  $(V_1, V_{Ins})$  where analogously,  $V_{Ins}$  is  $C_{Ins}(V_1)$ . In both cases we define two sets,  $I_{Del}$  and  $I_{Ins}$  of interferences. Each element in an interference set is a dependency triple extracted from the dependency set of the “potentially interfering” version, which causes  $NI_v$  to be not satisfied.

The interference set  $I$  for  $(V_1, V_2)$  is the union set of  $I_{Del}$  and  $I_{Ins}$  where each element is only taken once.

We now go back to the problem from which we started. In the context of this work we are interested in determining interfering deltas rather than interfering

versions. The conclusion that we reach on the condition of non-interference for deltas stems from the idea that *when two changes – no matter how separated in time – are found to be parallel and should therefore not interfere, the latter change should have no semantic impact on the program points affected by the former change*. When we check a new delta for possible interferences we ask that the modifications introduced do not affect the same program points affected by the earlier change in an interfering fashion.

	$V_1$	$V_2=V_3$	$V_4$
1:	program	program	program
2:	a:=1	a:=1	a:=1
3:	b:=2	b:=2	a:=2
4:	c:=a+b	b:=3	b:=2
5:	return c	c:=a+b	b:=3
6:	end	return c	c:=a+b
7:		end	return c
8:			end

Fig. 10 – Two deltas that are adjacent in time will only refer to three different versions since  $V_2$  and  $V_3$  will be identical.

Let us consider first the case of two parallel adjacent changes<sup>13</sup>. Fig. 10 shows two adjacent deltas for a simple program similar to the ones analyzed until now.

Using the technique discussed thus far we can determine  $\Delta_1$ ,  $\Delta_2$ ,  $A_1$ ,  $A_2$  and the interference sets  $I_1$  and  $I_2$ .

$$\Delta_1 = \{[a: 2 \mapsto 4], [b: 3 \mapsto 4], [c: 4 \mapsto 5]\}, \Delta_2 = \{[a: 2 \mapsto 5], [b: 4 \mapsto 5], [c: 5 \mapsto 6]\}$$

$$\Delta_3 = \{[a: 3 \mapsto 6], [b: 5 \mapsto 6], [c: 6 \mapsto 7]\}$$

$$A_1 = \{(1,1), (2,2), (\varepsilon,3), (3,4), (4,5)\}, A_2 = \{(1,1), (\varepsilon,2), (2,3), (3,4), (4,5), (5,6)\}$$

$$A_1 \bullet A_2 = \{(1,1), (\varepsilon,2), (2,3), (\varepsilon,4), (3,5), (4,6)\} \quad I_1 = \{[b: 4 \mapsto 5]\}, \quad I_2 = \{[a: 3 \mapsto 6]\}$$

We also compute the composition of  $A_1$  and  $A_2$  which transforms the program points of  $V_1$  in program points of  $V_4$  as if only one composed delta had been applied.

The generic element of an interference set, as discussed, is a variable dependency, i.e. a variable and its effective definition for a use made thereof. It therefore defines a reference to two program points, the one in which the variable is defined and the one in which the variable is used. A variable definition associated with a program point, as in this case, is a valid candidate for a slicing criterion. A *forward slice* of such a criterion will compute exactly the set of program points affected by such a definition, i.e. the set of program points corresponding to that part of code on which the change had impact.

Since in general a set of program points is produced for each interference, we define the set  $S$ , called *impact set* as

$$S = \bigcup_{i \in I} FS(V, i)$$

i.e. the union set of all the impact sets created by computing forward slices of a version  $V$  of the code, using each interference in an interference set as the slicing criteria. When impact sets are computed, the version that is being sliced is the version produced by the change. For our example:

<sup>12</sup> the only use in  $V_2$  is at program point 7 and (9,7) belongs to  $A$

<sup>13</sup> In this case the artificial delta  $\delta_2$  is irrelevant since  $V_2$  and  $V_3$  will be identical

$$S_1 = S_{\delta_1} = \bigcup_{i \in I_1} FS(V_2, i) \quad S_2 = S_{\delta_2} = \bigcup_{i \in I_2} FS(V_4, i)$$

The domains from which the elements of each set are drawn therefore are not the same. While the first impact set will be made of program points of  $V_2$ , the second impact set will accommodate program points of  $V_4$ . We define a particular composition of an  $A$ -application as

$$A(S_{\delta}) = \{p : \forall q_i \in S_{\delta}, (q_i, p_i) \in A\}$$

with proper extensions for composed applications.

The non-interference condition for adjacent deltas will thus be the following.

$$A(S_{\delta_1}) \cap S_{\delta_2} = \emptyset \quad (NI_{\delta})$$

i.e. the set of program points impacted by the original change and the set of program points affected by the second change ought to be separated.

### The General Case: Non-Adjacent Changes

Fig. 7 shows the flow graph for the general case of non-adjacent changes. In this case the number of versions involved grows to four, but we must also adjust the definition of semantic interferences between changes.

	$V_1$		$V_2$
	<b>program</b>		<b>program</b>
1:	a:=2		a:=2
2:	b:=&a		b:=&a
3:	c:=a+2		a:=3
4:	d:=*b		c:=a+2
5:	e:=c*d	←	d:=*b
6:	write e		e:=c*d
7:	end		write e
8:			end
	$V_3$		$V_4$
	<b>program</b>		<b>program</b>
1:	a:=2		a:=2
2:	b:=&a		b:=&a
3:	a:=3		a:=3
4:	c:=a+2		a:=4
5:	c:=4	←	c:=a+2
6:	d:=*b		c:=4
7:	e:=c*d		c:=5
8:	write e	←	d:=*b
9:	end		e:=c*d
10:			write e
11:			end
12:			

Fig. 11 - Two generic deltas involve four different versions of code. In general, the differences between the intermediate versions will be consistent, especially when the deltas are greatly separated in time. In the example,  $\delta_1$  inserts two lines and while both manifestly interfere with  $V_3$ , only the insertion of line 5 interferes with the changes made in  $\delta_2$ .

When two deltas are not adjacent, the intermediate versions  $V_2$  and  $V_3$  will usually be very different. Specifically,  $V_3$  will often interfere semantically with  $V_4$  accordingly to  $NI_v$  because of the many deltas that may not have been parallel in the original context and that occurred in the lapsed time. Since the latter delta, which must be guaranteed not to interfere with the former delta, starts from a consolidated version  $V_3$ , it is reasonable to assume that the interferences introduced by  $V_3$  over  $V_2$  have been tested, accepted and integrated in the system.

These interferences, independently from whether or not they conflict with the intended changes made in  $\delta_1$ , should be intentionally prevented from being the trigger of a positive assessment of interference between  $\delta_1$  and

$\delta_2$ . With this rationale in mind, we define an operation between interference set: the **variable subtraction** operation between interference sets, indicated by  $\otimes$  is therefore defined as follows.

Let  $I_1$  and  $I_2$  be two interference sets.<sup>14</sup>

$$I_1 \otimes I_2 = \{[v : d \mapsto u] : [v : d \mapsto u] \in I_1 \wedge (\forall (d_2, u_2), [v : d_2 \mapsto u_2] \notin I_2)\}$$

An example will greatly simplify the understanding of this very simple operation.

$$\begin{aligned} I_1 &= \{[a : 1 \mapsto 3], [a : 4 \mapsto 5], [b : 2 \mapsto 3], [b : 6 \mapsto 7], [c : 7 \mapsto 8]\} \\ I_2 &= \{[b : 2 \mapsto 8], [b : 9 \mapsto 10], [b : 9 \mapsto 12], [c : 6 \mapsto 7], [d : 8 \mapsto 9]\} \\ I_1 \otimes I_2 &= \{[a : 1 \mapsto 3], [a : 4 \mapsto 5], I_2 \otimes I_1 = \{[d : 8 \mapsto 9]\} \end{aligned}$$

In other words the operation deletes from the first interference set the dependencies defined over variables for which some dependency is also defined in the second operand.  $I_2$ , in the example above, has dependencies defined for  $b$  and  $c$  and so does  $I_1$ . Therefore the dependencies relative to these variables are removed from  $I_1$ . The example also shows clearly that this operation is not commutative.

As it is shown in the flow chart of Fig. 7, the process of detecting semantic interferences between generic changes is a three step process. First, three interference sets are determined. These three sets represent the semantic interferences introduced by  $\delta_1$  and  $\delta_2$  by modifying  $V_1$  and  $V_3$  and the set of interferences introduced by the combination  $\delta_2$  of all changes submitted between  $\delta_1$  and  $\delta_2$ .

In a second step, the set of interferences caused by  $\delta_2$  is  $\otimes$ -filtered of those interferences which overlap with interferences introduced by any of the intermediate deltas.

$$I_2 = I_2 \otimes I_e$$

By means of this operation we are indeed avoiding to look at those variables whose dependencies have already been affected by some change which occurred before  $\delta_2$ . Such changes have been submitted at some point in the past and yet they have been assimilated in the version history and accepted (admittedly, we do not know whether they interfered or not with  $\delta_1$  and if they did, they so did with good reason); therefore we argue that any further change that affects semantically what has already been affected by some  $\delta_e$  cannot constitute a semantic interference of  $\delta_2$  on  $\delta_1$ .

Applying this procedure to the example of Fig. 11, we compute the four dependency sets,

$$\begin{aligned} \Delta_1 &= \{[a : 1 \mapsto 3], [a : 2 \mapsto 4], [a : 2 \mapsto 5], [b : 3 \mapsto 5], [c : 4 \mapsto 6], [d : 5 \mapsto 6], [e : 6 \mapsto 7]\} \\ \Delta_2 &= \{[a : 1 \mapsto 3], [a : 4 \mapsto 5], [a : 4 \mapsto 6], [b : 3 \mapsto 6], [c : 5 \mapsto 7], [d : 6 \mapsto 7], [e : 7 \mapsto 8]\} \\ \Delta_3 &= \{[a : 1 \mapsto 3], [a : 4 \mapsto 5], [a : 4 \mapsto 7], [b : 3 \mapsto 7], [c : 6 \mapsto 8], [d : 7 \mapsto 8], [e : 8 \mapsto 9]\} \\ \Delta_4 &= \{[a : 1 \mapsto 3], [a : 5 \mapsto 6], [a : 5 \mapsto 9], [b : 3 \mapsto 9], [c : 8 \mapsto 10], [d : 9 \mapsto 10], [e : 10 \mapsto 11]\} \end{aligned}$$

<sup>14</sup> This formula reads as following: the variable subtraction set between two interference sets is an interference set whose members are members of the first operand, for which there are no members of the second operand set that specify a dependency for the same variable.



and the three  $A$ -applications that transform program points from one version to the next,

$$\begin{aligned} A_1 &= \{(1,1), (2,2), (3,3), (e,4), (4,5), (5,6), (6,7), (7,8)\} \\ A_e &= \{(1,1), (2,2), (3,3), (4,4), (5,5), (e,6), (6,7), (7,8), (8,9)\} \\ A_2 &= \{(1,1), (2,2), (3,3), (4,4), (e,5), (5,6), (6,7), (e,8), (7,9), (8,10), (9,11)\} \end{aligned}$$

The four dependency sets show some particularities. The different versions of the program of Fig. 11 show for the first time in our analysis the use of pointers. We mentioned how pointers and pointer analysis affect the classification of semantic interferences and how special operations with these objects greatly affect the degree of side-effects that can be manifest in code and the difficulty, therefore, of detecting interferences. If we look at  $V_1$  as the version we refer to, we notice that the variable  $a$  has a dependency from program point (1) to program point (3), in which variable  $b$  is defined as a pointer to  $a$ . In the model we present, we use program point (1) to indicate the allocation of local variables on the stack along with their respective memory addresses. What is used at line 3 therefore is a memory address that was defined at such a program point, in order to be assigned to  $b$ . Under the same assumptions, we have two further dependencies, for  $a$  and for  $b$ , from program points (2) and (3) respectively, to program point (5). At line 5, only variable  $b$  is explicitly mentioned. Nonetheless, by dereferencing  $b$ , a use of both its direct value (defined at (3)) and indirect value (the value of  $a$ , defined at (2)) is made. This particularity is present in all further versions and is a characteristic of how we describe dependencies when pointers are involved.

Following the flow diagram of Fig. 7 and the process that we described this far, we compute the three interference sets,  $I_1$ ,  $I_e$ , and  $I_2$ .

$$I_1 = \llbracket a : 4 \mapsto 5 \rrbracket \llbracket a : 4 \mapsto 6 \rrbracket, I_e = \llbracket c : 6 \mapsto 8 \rrbracket, I_2 = \llbracket a : 5 \mapsto 6 \rrbracket \llbracket a : 5 \mapsto 9 \rrbracket \llbracket c : 8 \mapsto 10 \rrbracket$$

We can easily see that the cumulative change  $\delta_e$  from  $V_2$  to  $V_3$  introduces a semantic interference on variable  $c$ . The potentially interfering delta in our inspection process ( $\delta_2$ ) also interferes on variable  $c$ . Such interference should not be considered a possible cause of conflict between  $\delta_2$  and  $\delta_1$  as  $c$  has been already interfered upon by changes that – one way or the other – have been successfully integrated in the version history. Following once again the process described in Fig. 7 we  $\otimes$ -subtract the set of intermediate interferences from the interferences introduced by  $\delta_2$ .

$$I_2 = I_2 \otimes I_e = \llbracket a : 5 \mapsto 6 \rrbracket \llbracket a : 5 \mapsto 9 \rrbracket$$

Looking at  $\delta_2$  and at the modifications it makes to the code, and considering the previous versions altogether, we recognize that while the insertion of  $a:=4$  disrupts the intended flow on the original data successors of  $a:=3$ , the same cannot hold for the interference on the variable  $c$ , as the data flow on this variable's definition's successors has already been disrupted by the cumulative change  $\delta_e$ . If we

were trying to investigate the interferences between  $\delta_2$  and that particular change that originally changed the data flow for  $c$  we might be likely to reach the conclusion that the two deltas do indeed interfere; nonetheless in our particular case, the effects of  $\delta_e$  have made so that the impact of  $\delta_2$  on  $\delta_1$  is reduced. We indeed expect our technique to show that, only the addition of  $a:=4$  introduced a semantic conflict with  $\delta_2$ .

We now ought to compute the impact that each interference set has on the global system dependency graph. The first impact set,  $S_1$  is created as the union set of the forward slices of  $V_2$  when each interference in  $I_1$  is used as a slicing criterion.

<pre> 1: program 2: a:=2 3: b:=&amp;a 4: a:=3 5: c:=a+2 6: d:=*b 7: e:=c*d 8: write e 9: end </pre>	→	$I_1 = \llbracket a : 4 \mapsto 5 \rrbracket \llbracket a : 4 \mapsto 6 \rrbracket$ $S_1 = \{5,6,7,8\}$
---	---	--

In this case, since the two interferences are related to the same definition program point, the impact set and the only forward slice computed, coincide. In general, a set of program points for each interference set will be produced.

Similarly for  $S_2$ ,

<pre> 1: program 2: a:=2 3: b:=&amp;a 4: a:=3 5: a:=4 6: c:=a+2 7: c:=4 8: c:=5 9: d:=*b 10: e:=c*d 11: write e 12: end </pre>	→	$I_2 = \llbracket a : 5 \mapsto 6 \rrbracket \llbracket a : 5 \mapsto 9 \rrbracket$ $S_2 = \{6,9,10,11\}$
--	---	--

Naturally, the example that we are discussing is a very simple one. In general though, the slicing operation will reveal more subtle data dependencies and less obvious impact sets. As extensions to this work we plan to adapt and analyze more complicated and interesting examples. It should be easy to imagine though that the interferences introduced by two deltas can be completely unrelated and the respective impact sets completely separated. In this case we reckon that the two changes do not conflict with each other. In our example though,

$$\begin{aligned} A &= A_e \cdot A_2 = \{(1,1), (2,2), (3,3), (4,4), (e,5), (5,6), (e,7), (e,8), (6,9), (7,10), (8,11)\} \\ A(S_1) &= \{6,9,10,11\}, S_2 = \{6,9,10,11\}, A(S_1) \cap S_2 = \{6,9,10,11\} \neq \emptyset \end{aligned}$$

The two impact sets are not separated – in our example, as a matter of fact they completely overlap – showing that the considered interferences have impact on common program points. We decide that  $\delta_2$  interferes with  $\delta_1$  under the assumptions and definitions used this far.

## Optimistic CMS and Concurrent Changes

While two deltas that are submitted for integration in a concurrency situation are simpler to classify as parallel, they also are rather counter-intuitive to define in terms of inter-deltas interference.

While being not simpler the sequential one, the concurrent case is not particularly harder. There still are four versions involved: *the original version, two concurrent versions produced by each delta, and a unified version produced by a chosen merging algorithm*. These versions are not linearly ordered on a time scale, and four deltas ( $\delta_1, \delta_2, \delta_1^*, \delta_2^*$ ) govern the variations between the versions.

First, when concurrent changes are made to the code, a merging strategy is necessary. At any given time there ought to be only one consistent version of the code which is allowed to be modified, and all modifications stemming from the same reference version are considered concurrent.

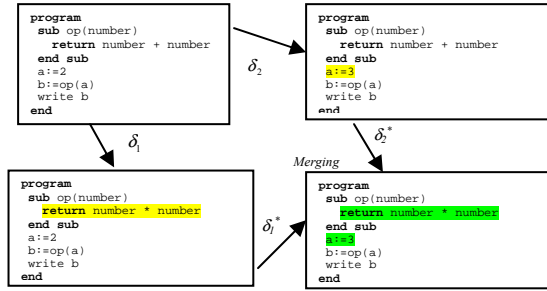


Fig. 12 – Merging versions which do not present local conflicts but affect each other impact slices, may produce a version that defies the original changes or conflicts with the behavior expected by each or either one of the developers that submitted the changes.

Several merging strategies and algorithms have been proposed [9] and while some involve looking at syntactic conflicts and slicing for delineating local interferences in the program flow, they produce integrated versions which might include elements that defy the original intents of either one or both of the authors of the concurrent versions. Fig. 12 shows an example of such a phenomenon.

We believe that deciding whether or not two concurrent changes interfere is equivalent to deciding whether or not the merged version interferes semantically with them.

The last flow block of the rightmost branch of Fig. 7 incorporates this duality. We argue that  $\delta_1$  interferes with the concurrent change  $\delta_2$  if and only if the respective changes necessary to construct the merged version starting from  $V_1$  and  $V_2$  interfere semantically with either one of the original deltas.

In symbols,  $\delta_1$  and  $\delta_2$  are concurrent changes,  $\delta_1^*$  and  $\delta_2^*$  are the changes that bring each concurrent version in the merged one, and the symbol  $\triangleleft$  indicates semantic interference (see Fig. 12),

$$\delta_1 \triangleleft \delta_2 \Leftrightarrow (\delta_1^* \triangleleft \delta_1) \vee (\delta_2^* \triangleleft \delta_2)$$

Note that while the leftmost part of the above expression is a commutative interference between

concurrent changes, the interference relations contained in the rightmost part of the equivalence are non commutative interferences between adjacent time-ordered changes.

One could easily verify that the process sketched in Fig. 7 realizes the equivalence above. In doing so, one should remember that  $A_I^*$  is the application that links program points of  $V_i$  to program points of the merged version  $V_U$ .

In [19] we delineate the design of the Semantic Conflict Analyzer (SCA), a development aid in large scale SW development that interoperates with the version management system in use. Its general architecture is shown in Fig. 13.

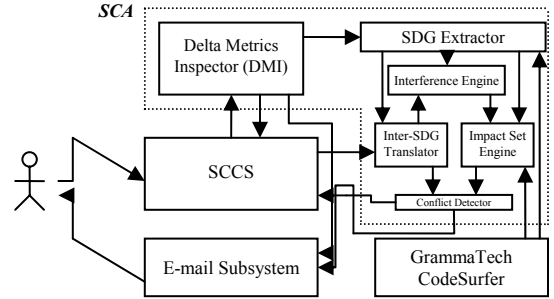


Fig. 13 – The component architecture of the Semantic Conflict Analyzer. In its reference design the SCA is interfaced with SCCS, CodeSurfer, and a generic email subsystem.

## Conclusion

In this paper, we showed how parallel changes may interfere semantically, subverting intended modifications and causing unintended behavior. We described some ways in which software changes may interact and interfere at a semantic, rather than syntactic level, and we showed that semantic interferences derive from intrusions in the established data flow of an execution slice, through *def-use overlapping*. We then described a technique based on code slicing and dataflow analysis to detect semantic conflicts among parallel changes. We extended the notion of semantic interferences between versions of source code to that of conflicts between changes and we defined a formal notion of parallel changes as modifications to the code that work against the intent of the author of the original modification.

For future directions of this research, we plan to build a prototype of SCA [19], perform empirical studies to evaluate the efficacy of our technique and its performance under different sets of environmental conditions and constraints, and use the prototype to determine the extent of both interfering and non-interfering parallel changes.

## Bibliography

- [1] D. E. Perry, H. P. Siy and L.G. Votta. *Parallel Changes in Large-Scale Software Development: An*

- Observational Case Study*, ACM Transactions on Software Engineering and Methodology, 10:3 (July 2001), 308-337.
- [2] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J.S. Marron, and Audris Mockus. *Does Code Decay? Assessing the Evidence from Change Management Data* in IEEE Transactions on Software Engineering, Vol. 27, No. 1, January 2001
  - [3] Frank Tip. *A survey of program slicing techniques*, Journal of Programming Languages 3 (1995) 121-189, 1995
  - [4] P. Anderson, and T. Teitelbaum. *Software Inspection Using CodeSurfer*. Workshop on Inspection in Software Engineering (CAV 2001), Paris, France., July 18-23, 2001.
  - [6] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. IEEE Computer pages 10 19, April 1987
  - [7] David B. Leblang. *The {CM} Challenge: Configuration management That Works* in Walter F. Tichy, editor, Configuration Management. Trends in Software, John Wiley and Sons, 1994
  - [8] Gregor Kiczales, John Lamping, Anurag Menhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. *Aspect-Oriented Programming*. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Springer-Verlag LNCS 1241. June 1997
  - [9] Susan Horwitz, Jan Prins, Thomas Reps. *Integrating Noninterfering Versions of Programs* ACM Transactions on Programming Languages and Systems, Vol. 11, No. 3, July 1989, Pages 345-387
  - [10] Ulf Ask Lund and Boris Magnusson. *A Case-Study of Configuration Management with ClearCase in an Industrial Environment*. System Configuration Management, 1997.
  - [11] M.M. Lehman, D.E. Perry and J.F. Ramil. *Implications of Evolution Metrics on Software Maintenance*. ICSM'98, November 1998.
  - [12] Ranjith Purushothaman, Dewayne E. Perry, *Towards Understanding Software Evolution: One-Line Changes*, To appear in 2002 International Conference on Software Engineering, Portland, June 2002
  - [13] Dewayne E. Perry, Harvey P. Siy, *Challenges in Evolving a Large Scale Software Product*, Proceedings of the International Workshop on Principles of Software Evolution, 1998 International Software Engineering Conference, Kyoto, Japan, April 1998
  - [14] Audris Mockus, Lawrence G. Votta, *Identifying Reasons for Software Changes using Historic Databases*, in International Conference on Software Maintenance, San Jose, California, October 14, 2000, Pages 120-130
  - [15] Yamin Wang, Wei-Tek Tsai, Xiaoping Chen, Sanjai Rayadurgam, *The role of Program Slicing in Ripple Effect Analysis*, 8<sup>th</sup> International Conference on Software Engineering and Knowledge Engineering, 1996, 369-376.
  - [16] Tom Mens, *A State-of-the-art Survey on Software Merging*, IEEE Transactions on Software Engineering, Vol. 28, No. 5, May 2002
  - [17] GrammaTech Inc, *CodeSurfer User Guide and Technical Reference*.
  - [18] Wu Yang, Susan Horwitz and Thomas Reps, *A Program Integration Algorithm that Accommodates Semantics-Preserving Transformations*, ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 3, July 1992, Pages 310-354
  - [19] G. Lorenzo Thione, *Detecting Semantic Conflicts in Parallel Changes*, Master's Thesis, University of Texas at Austin, December 2002