DELTAIMPACTFINDER: Assessing Semantic Merge Conflicts with Dependency Analysis

Martín Dias Guillermo Polito Damien Cassou Stéphane Ducasse

RMoD
Inria Lille–Nord Europe — University of Lille — CRIStAL, France

Abstract

In software development, version control systems (VCS) provide branching and merging support tools. Such tools are popular among developers to concurrently change a codebase in separate lines and reconcile their changes automatically afterwards. However, two changes that are correct independently can introduce bugs when merged together. We call *semantic merge conflicts* this kind of bugs.

Change impact analysis (CIA) aims at estimating the effects of a change in a codebase. In this paper, we propose to detect semantic merge conflicts using CIA. On a merge, DELTAIMPACTFINDER analyzes and compares the impact of a change in its origin and destination branches. We call the difference between these two impacts the *delta-impact*. If the delta-impact is empty, then there is no indicator of a semantic merge conflict and the merge can continue automatically. Otherwise, the delta-impact contains what are the sources of possible conflicts.

1. Introduction

Software projects are in constant evolution. Often, developers perform changes concurrently in a codebase, generating separate lines of development. Version control systems (VCS) support this activity through branches, a widely used feature [6, 13] in software development. Merge [10] (also called integration) is a fundamental operation in VCS that reconciles two (or more) branches. VCS can detect syntactical merge conflicts automatically. Nevertheless, semantic merge conflicts (*i.e.,* at the level of program behavior) exceed the scope of these tools. Consider, for example, a branch renaming a template method from A»foo to A»bar and another branch overriding foo in a B class, subclass of A. The two branches can be automatically merged but the resulting code will fail to execute as intended: indeed B»foo will never be executed while B»bar is supposed to exist but does not, Such *semantic merge conflicts* are not detected by current VCS.

Change Impact Analysis [3, 9] (CIA) is an active research field that aims at identifying the potential consequences of a change in a codebase. Typically, CIA techniques establish *dependency* relationships between the code entities of the codebase. These relationships are used afterwards for detecting the set of entities that are impacted by a change. The rationale is that when a code entity changes, the behavior of the *dependent* entities is impacted. Many CIA research works use the technique of computing the dependencies of a change in the original codebase where the authors created the change [1, 5, 7, 12].

This paper proposes a solution to help integrators in the detection of semantic merge conflicts using CIA. On a merge, DELTAIMPACTFINDER analyzes and compares the impact of a change in its origin and destination branches. We call the difference between these two impacts the *delta-impact*. If the delta-impact is empty, it means that there is no semantic merge conflict and the merge can continue automatically. Otherwise, the delta-impact contains what are the sources of possible conflicts. The contributions of this paper are the following:

- a description of semantic merge conflict with an example (Section 2);
- a CIA technique, named DELTAIMPACTFINDER, to detect semantic merge conflicts (Section 3);
- a discussion of usages of this technique (Section 4);
- a prototype of this technique implemented in Pharo (Section 5);

2. Problems when Merging: Semantic Conflicts

To show how semantic conflicts appear when merging, we start by introducing an example of the Fragile Base Class Problem [11] (FBCP). Consider the following logging library that implements a Log class whose API has the methods log:, which can record a single message into an internal collection, and logAll:, which records multiple messages in one shot using log:. The logic for adding an element to the collection of logs is only expressed inside the log: method. Following there is the code illustrating the most relevant points of such implementation:

```
Object subclass: #Log
  instanceVariableNames: 'messages'.

Log ≫ log: aMessage
  messages add: aMessage.

Log ≫ logAll: someMessages
  someMessages do: [:each | self log: each ].
```

We want now to introduce a change in this library. At some point in time, a developer starts a new branch of the library from version A and implements a new feature: the FilteredLog (Figure 1). FilteredLog is a subclass of Log that overrides log: to record the message only when it satisfies a filter. Note that FilteredLog does not need to override Log≫logAll:. We refer to this change as Δ_F. The code illustrating Δ_F is the following:

```
+ Log subclass: #FilteredLog
+    instanceVariableNames: 'filterBlock'.
+
+ FilteredLog ≫ log: aMessage
+    (filterBlock value: aMessage)
+       ifTrue: [ super log: aMessage ].
```

In parallel, the main branch of the library evolves: the method Log≫logAll: no longer uses self log: to record each received message, but instead each message is added directly to the internal collection.

```
  Log ≫ logAll: someMessages
-    someMessages do: [:each | self log: each ].
+    messages addAll: someMessages
```

When the integrator wants to merge Δ_F in the main branch, the tool does not inform any merging conflicts but the introduced feature does not work as expected. Indeed, FilteredLog does not filter any messages when using logAll: because this method does not use the log: message anymore.

```
log := FilteredLog new.
log filterBlock: [ :each | each > 0 ].
log logAll: #(-5).
log messages isEmpty. "false ---> wrong!"
```

We can observe from this example that an integrator can merge Δ_F introducing a semantic conflict silently. To detect such kind of bugs, integrators need to understand the code in a change (Δ) more deeply *e.g.,* know the intention of the change, in which version was it developed. Then, the activity of integration requires a big human effort which new tools can help to alleviate.

In a more general way, we would like a tool that helps the integrator by answering the following questions:

Q1. *Does a Δ produce semantic conflicts if merged in a particular version of the codebase?*

Q2. *What code entities are involved in a semantic conflict that Δ produces?*

Q3. *When was the change that produced a semantic conflict with a Δ integrated?*

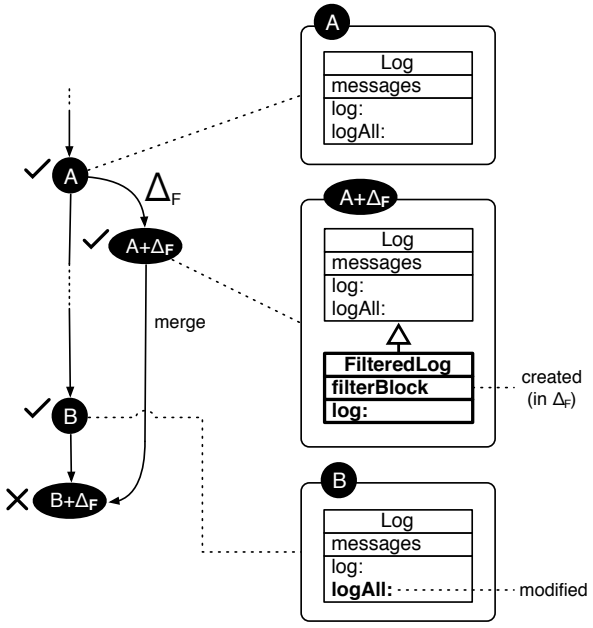

Figure 1. A developer starts a new branch of the library from version A of a logging library and implements a new feature: the FilteredLog. We name such change as Δ_F. Before Δ_F is integrated into the library, a modification in the method Log≫logAll: in B makes $B + \Delta_F$ not working.

In general these questions are hard to answer, specially in large scale projects where developers work in parallel on the same codebase. Automated Testing and Continuous Integration practices could help in answering the first question. However, these practices depend on the coverage of the testing: the more tested is the code, the more likely the problem can be detected. Unfortunately, sometimes test coverage is not good enough and requires an effort that developers do not make. Then, we pose the following research question:

Can CIA on origin and destination branches help to answer these questions?

3. Our Solution: DELTAIMPACTFINDER

In a nutshell, we propose to detect semantic merge conflicts using CIA. On a merge, DELTAIMPACTFINDER analyzes and compares the impact of a change in its origin and destination branches. We call the difference between these two impacts the *delta-impact*. If the delta-impact is empty, it means that there are no semantic conflicts and the merge can continue automatically. Otherwise, the delta-impact contains what are the sources of possible conflicts.

In the following we describe our approach. First, we define the notions of dependency and impact. Then, based in such notions, we explain delta-impact.

3.1 Dependency and Impact

Change Impact Analysis [3, 9] (CIA) aims at identifying the potential consequences of a change in a codebase. Typically, CIA techniques establish *dependency* relationships between the code entities of the codebase, which they use afterwards for detecting the set of entities that are impacted by a change. The rationale is that when a code entity changes, the behavior of the *dependent* entities is impacted. In the context of this paper, we define dependency as follows:

DEFINITION 1. *Dependency. A* dependency *is the relationship between two code entities where one code entity* $(source)$ *requires the other* $(target)$*. We denote it* $source \rightarrow target$*.*

In the motivational example we introduced in Section 2, the FilteredLog class depends on the Log class because of the inheritance relationship between them. In this paper we focus on static dependency analysis, *i.e.,* dependencies that are explicit in the source code, however we believe that DELTAIMPACTFINDER can be generalized to other kinds of dependencies. We describe the dependencies of DELTAIMPACTFINDER in more detail in Section 5.1.

Then, we define the impact of a Δ as follows:

DEFINITION 2. *Impact. The* impact *of a change* Δ *in a codebase* C*, denoted* $I(\Delta, C)$*, is the set of dependencies introduced or removed in* C *after applying* Δ*.*

In the motivational example, the impact of Δ in its origin branch includes the following dependency modifications (Figure 2):

- i_1 Introduction of an inheritance dependency from FilteredLog to Log.
- i_2 Introduction of a message send dependency from Log»logAll: to FilteredLog»log:.
- i_3 Introduction of a message send dependency from FilteredLog»log: to Log»log:.

3.2 DELTAIMPACTFINDER

In the example (Figure 3), the comparison of the impact of Δ_F in origin and destination branches shows that the dependency from Log»logAll: to FilteredLog»log: is missing in the destination branch. Precisely, the cause of the semantic conflict in the example is the change in Log»logAll:, which no longer invokes FilteredLog»log:.

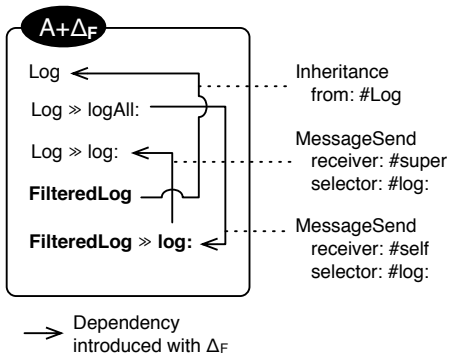

Figure 2. Impact of Δ_F in A. In the origin branch, Δ_F introduces three dependencies: one corresponding to an inheritance relationship, and the others corresponding to message sends.

We observe that the impact of a Δ contains a set of relationships between code entities in a particular version of code. Informally, we can think about this impact as a set of constraints that have to be satisfied for the code to work as expected. Then, we pose the following hypothesis:

A semantic conflict appears when the set of dependencies that a Δ *introduces (or removes) into its origin branch is different from those introduced (or removed) when merging such* Δ *in the destination branch.*

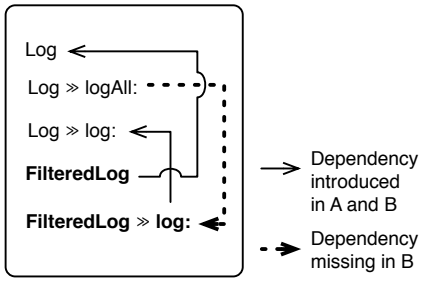

Figure 3. Delta-impact of Δ_F from A to B. Comparison of the impact of Δ_F in both A and B. The dependency from Log»logAll: to FilteredLog»log: is missing in B.

In other words, if there are missing or extra dependencies in the destination branch, this could mean that Δ has a different meaning than the one intended by its author. On the contrary, if the dependencies are the same the change may have the same effects. Then, we define *delta-impact* as follows:

DEFINITION 3. *Delta-Impact. The* delta-impact *of a change* Δ *with origin branch* A *and destination branch* B*, denoted*

$DI(\Delta, A, B)$, is the symmetric difference[1] between the set of impacts of Δ in A and the impact of Δ in B.

To compute the delta-impact of Δ, we obtain the impact of Δ in the origin and destination branches, and then we compute the symmetric difference between the two impacts. In the example (Figure 4), the impact $I(\Delta_F, A)$ yields the set $\{i_1, i_2, i_3\}$, while $I(\Delta_F, B)$ yields the set $\{i_1, i_3\}$. Then, $DI(\Delta_F, A, B)$ results in the set $\{i_2\}$, because i_2 is missing in the destination branch B. In this context, a tool that computes and shows the delta-impact of Δ_F to the integrator would answer the questions Q1 and Q2 that we defined in Section 2.

$I(\Delta_F, A)$:

+	FilteredLog	→	Log	(i_1)
+	Log»logAll:	→	FilteredLog»log:	(i_2)
+	FilteredLog»log:	→	Log»log:	(i_3)

$I(\Delta_F, B)$:

+	FilteredLog	→	Log	(i_1)
+	FilteredLog»log:	→	Log»log:	(i_3)

$DI(\Delta_F, A, B) = I(\Delta_F, A) \ominus I(\Delta_F, B)$:

+	Log»logAll:	→	FilteredLog»log:	(i_2)

Figure 4. DELTAIMPACTFINDER $I(\Delta_F, A)$ is the original impact of Δ_F, while $I(\Delta_F, B)$ is the destination impact. $DI(\Delta_F, A, B)$ is the delta-impact of Δ_F in the destination branch, which shows that a dependency introduction is missing (i_2).

4. Applicability

In this section we describe concrete scenarios where we would like to validate DELTAIMPACTFINDER (in future works). Since our plan is to validate our approach with Pharo community, we present these scenarios in terms of Pharo.

4.1 Aged Code Changes

The codebase of a large open-source project like Pharo evolves through code changes that developers submit. When a developer submits a code change, this change must pass a reviewing process before an integrator merges it into the main development branch. Since this reviewing process takes some time and code changes are integrated every day, when the integrator has to merge an approved code change, the current Pharo codebase may be different than the codebase where the author of the change worked.

In this situation, DELTAIMPACTFINDER can help Pharo integrators to discover semantic conflicts on the merge. Given a code change Δ submitted by a developer, DELTAIMPACTFINDER can answer:

"Is the impact of Δ in the current Pharo the same as the impact in the Pharo where Δ was originally created?"

4.2 Software Migration

Often, when the codebase of a project changes, other projects that depend on it need to be migrated. This change propagation is known as ripple effect [20]. Ripple effects are problematic because a small change in a project can have a very large impact on other projects. Additionally, sometimes a code that needs migration remains undiscovered for a long time due to low test coverage.

We can illustrate this scenario by rephrasing the FBCP example used in Section 2. Let's suppose that a developer works in a project in Pharo version A. The system provides the class Log, which the developer extends in his project by creating the subclass FilteredLog. One year later, a new stable version of Pharo is available: version B. Among plenty of changes in Pharo B, the method Log»logAll: has been modified like in the FBCP example. As before, a bug appears in the method logAll: when invoked in an instance of FilteredLog. Note that when the developer loads the FilteredLog class in Pharo B, the system does not raise any load or compilation error: it is another form of the semantic merge conflict. The developer will probably have to debug his project to find that the change in Log»logAll: is the responsible. If a tool would have informed the developer that Log»logAll: changed, then he could save time.

We can pose this problem in terms of our approach. When the FilteredLog package is loaded in Pharo A, some dependencies are introduced between code entities of FilteredLog and Pharo. This is what we have defined as impact. When the package is loaded in Pharo B, the impact is different: i_2 is missing (Figure 4). In general terms, DELTAIMPACTFINDER can help developers to answer the following question:

"Which Pharo code entities with impact on my project changed from Pharo A to B?"

4.3 Requirements

The main requirement of any CIA technique is a high precision and a high recall [9]. A high precision means that a technique finds substantially more relevant results than irrelevant, while high recall means that a technique finds most of the relevant results.

However, we extract some additional requirements for the implementation of DELTAIMPACTFINDER from the scenarios presented above in this section:

Isolation from tool's environment. For supporting "Software Migration", the implementation needs to compute dependencies of code entities as if they were loaded in some arbitrary Pharo version, independently of the Pharo version where the tool is actually running.

Usable in real use cases. Since we aim at building a tool that real developers can evaluate, the implementation should compute the dependencies in a reasonable time.

[1] the symmetric difference between two sets includes only the elements that belong to one of such sets but not to both.

5. Implementation

We start this section doing an overview of the main characteristics of our prototype implementation of DELTAIMPACTFINDER. Some design decisions are consequence of the requirements stated in Section 4.3.

Static code analysis. We use default Pharo support for performing static code analysis. For example, the AST-Core package provides support for visiting the *abstract syntax tree* of methods and collecting dependencies.

Light-weight and polymorphic code metamodel. We implemented RingFicus, a metamodel for Pharo code entities. It provides first-class representations for class, method, instance variable, etc. RingFicus allows to model code either internal or external to the current Pharo environment to browse them, query them, analyze them, as if they were loaded in the system.

5.1 Computing Dependencies

DELTAIMPACTFINDER requires to compute dependencies between source code entities. In our solution such entities are classes, metaclasses, instance and class variables, traits, class-traits and methods. The relationships that we consider as dependencies in DELTAIMPACTFINDER are the following:

Inheritance: A dependency from a class to its superclass.

Trait Usage: A dependency from a class, metaclass, trait, or class-trait to all traits in its *trait composition*.

Variable Access: A dependency from a method in a class or metaclass that accesses (read or write) an instance or class variable, to the accessed variable.

Message Send: A dependency from a method including a message-send sentence, to all the possible methods that are invoked. Due to the absence of type information in the language, in the general case the algorithm uses the selector of a message-send to look up for all the implementors in the codebase. However, in the case of self-sends and super-sends the algorithm refines its look up for obtaining more precise dependencies.

For testing our prototype, we implemented DependencyMiner, which iterates over all the source code entities of a Pharo environment collecting the dependencies. Each dependency is an association $source \rightarrow target$.

5.2 Computing Impact and Delta-Impact

For the computation of the impact of Δ in an environment (Figure 5), the algorithm starts by building the codebase $C + \Delta$. Then, the prototype computes the dependency sets of each environment using the DependencyMiner, described above. Finally, the prototype the impact and the delta-impact by performing Collection operations.

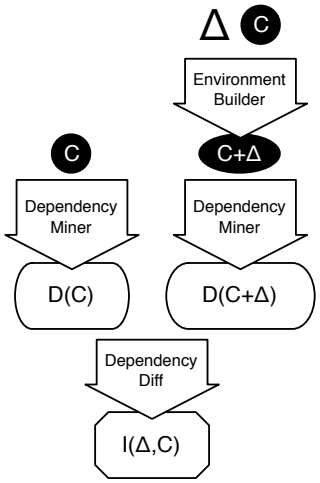

Figure 5. Computation of the impact of Δ in C. The algorithm starts by building the codebase $C + \Delta$. After, the algorithm computes the dependency sets of each codebase ($D(C)$ and $D(C + \Delta)$). Finally, the algorithm computes the impact by finding the symmetric difference between $D(E)$ and $D(E + \Delta)$.

6. Related Work

Change Impact Analysis. In an exhaustive survey [9] about CIA, Li *et al.* analyze 30 publications from 1997 to 2010 and identified 23 code-based CIA techniques. The study characterizes the CIA techniques, and identifies key applications of CIA techniques in software maintenance. Typically, CIA techniques establish *dependency* relationships between the code entities of the codebase, which they use afterwards for detecting the set of entities that are impacted by a change. The rationale is that when a code entity changes, the behavior of the *dependent* entities is impacted. The techniques to identify dependencies in a codebase are typically classified into static and dynamic. The static techniques identify the dependencies using static code analysis [14, 19], while dynamic techniques [2, 8] collect data from program execution. There are, as well, mixed techniques [4] which combine both techniques. DELTAIMPACTFINDER is orthogonal to the technique to identify dependencies, besides our prototype works with a static CIA technique.

Merging. The semantic merge conflicts have been studied before under different names. Mens [10] describes this problem in his survey of code merging. In this work, the author remarks that most approaches to software merging have been validated on imperative programming languages and it is not trivial to port these approaches to the object-oriented paradigm, due to late binding and polymorphism in object-oriented programming languages.

Ring Metamodel. Ring [17][16] is a source code metamodel that serves as a unified infrastructure for building tools in Pharo. While Ring has proven efficacy for dependency analysis tools [15, 18], we found some limitations that driven us to reimplement our own RingFicus package. In our early tests, Ring did not fulfill the requirements we described in Section 4. It did not ensure isolation from tool's environment, and it was not efficient for importing a while Pharo metamodel.

7. Conclusion and Future Perspectives

In this paper, we proposed a solution to help integrators in the detection of semantic merge conflicts using CIA. On a merge, DELTAIMPACTFINDER analyzes and compares the impact of a change in its origin and destination branches. We call the difference between these two impacts the *delta-impact*. If the delta-impact is empty, it means that there is no semantic merge conflict and the merge can continue automatically. Otherwise, the delta-impact contains what are the sources of possible conflicts.

In short, this paper makes the following contributions:

- a description of semantic merge conflict with an example;
- a CIA technique, named DELTAIMPACTFINDER, to detect semantic merge conflicts;
- a discussion of concrete scenarios to validate this technique in future work;
- a prototype of this technique implemented in Pharo.

Acknowledgements

We would like to thank Guille Polito, Santiago Bragagnolo and Lucas Godoy for their support during this work. This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council.

References

[1] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, and Andrea De Lucia. Identifying the starting impact set of a maintenance request: A case study. In *European Conference on Software Maintenance and Reengineering (CSMR 2000)*, pages 227–230, 2000.

[2] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 432–441, New York, NY, USA, 2005. ACM.

[3] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[4] Haipeng Cai and Raúl A. Santelices. A framework for cost-effective dependence-based dynamic impact analysis. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 231–240, 2015.

[5] Gerardo Canfora and Luigi Cerulo. Impact analysis by mining software and change request repositories. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, METRICS '05, pages 29–, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Martin Dias, Verónica Uquillas-Gomez, Damien Cassou, and Stéphane Ducasse. Software integration questions: A quantitative survey. Technical report, INRIA Lille, 2014.

[7] Ahmed E. Hassan and Richard C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Softw. Engg.*, 11(3):335–367, 2006.

[8] Bogdan Korel and Juergen Rilling. Dynamic program slicing methods. *Information & Software Technology*, 40(11-12):647–659, 1998.

[9] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 613–646, 2013.

[10] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[11] Leonid Mikhajlov and Emil Sekerinski. A Study of the Fragile Base Class Problem. In *Proceedings of the European Conference on Object-Oriented Programming*, 355–383. Springer-Verlag, 1998.

[12] Maksym Petrenko and Václav Rajlich. Variable granularity for improving precision of impact analysis. In *ICPC*, pages 10–19. IEEE Computer Society, 2009.

[13] Shaun Phillips, Jonathan Sillito, and Rob Walker. Branching and merging: An investigation into current version control practices. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pages 9–15, New York, NY, USA, 2011. ACM.

[14] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. Change impact analysis based on a taxonomy of change types. In *Proceedings of the 34th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2010, Seoul, Korea*, pages 373–382, 2010.

[15] Verónica Uquillas Gómez. *Supporting Integration Activities in Object-Oriented Applications*. PhD thesis, Vrije Universiteit Brussel - Belgium & Université Lille 1 - France, 2012.

[16] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Meta-models and infrastructure for smalltalk omnipresent history. In *Smalltalks'2010*, 2010.

[17] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Ring: a unifying meta-model and infrastructure for Smalltalk source code analysis tools. *Journal of Computer Languages, Systems and Structures*, 38(1):44–60, April 2012.

[18] Verónica Uquillas Gómez, Stéphane Ducasse, and Andy Kellens. Supporting streams of changes during branch integration. *Journal of Science of Computer Programming*, 2014.

[19] M. Welser. Program slicing. *IEEE Transactions on Software Engineering*, pages 352–357, 1984.

[20] Stephen S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference*, pages 60–65. IEEE Press, 1978.