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 \texttt{Log} class whose API has the methods \texttt{log:}, which can record a single message into an internal collection, and \texttt{logAll:}, which records multiple messages in one shot using \texttt{log:]. The logic for adding an element to the collection of logs is only expressed inside the \texttt{log:} method.

Following there is the code illustrating the most relevant points of such implementation:

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

Log \rightarrow \texttt{log: \texttt{aMessage}}
messages add: \texttt{aMessage}.

Log \rightarrow \texttt{logAll: \texttt{someMessages}}
  someMessages do: [\texttt{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 \texttt{A} and implements a new feature: the \texttt{FilteredLog} (Figure 1). \texttt{FilteredLog} is a subclass of \texttt{Log} that overrides \texttt{log:} to record the message only when it satisfies a filter. Note that \texttt{FilteredLog} does not need to override \texttt{logAll:]. We refer to this change as \texttt{\Delta F}. The code illustrating \texttt{\Delta F} is the following:

```
+ Log subclass: #FilteredLog
+  instanceVariableNames: 'filterBlock'.
+  filterBlock:
+    \texttt{[ :each | each > 0]}. \\
+  \texttt{ifTrue: [super log: aMessage]}. \\

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

```
Log \rightarrow \texttt{logAll: \texttt{someMessages}}
  someMessages do: [\texttt{each | self log: each}]. \\
  \texttt{messages addAll: \texttt{someMessages}}
```

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

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

We can observe from this example that an integrator can merge \texttt{\Delta F} introducing a semantic conflict silently. To detect such kind of bugs, integrators need to understand the code in a change (\texttt{\Delta}) 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:

- \textbf{Q1. Does a \Delta produce semantic conflicts if merged in a particular version of the codebase?}
- \textbf{Q2. What code entities are involved in a semantic conflict that \Delta produces?}
- \textbf{Q3. When was the change that produced a semantic conflict with a \Delta integrated?}

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:

\textit{Can CIA on origin and destination branches help to answer these questions?}
3. Our Solution: DELTA.ImpactFinder

In a nutshell, we propose to detect semantic merge conflicts using CIA. On a merge, DELTA.ImpactFinder 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 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 \( \text{source} \rightarrow \text{target} \).

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 DELTA.ImpactFinder can be generalized to other kinds of dependencies. We describe the dependencies of DELTA.ImpactFinder in more detail in Section 5.1.

Then, we define the impact of a \( \Delta \) as follows:

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

In the motivational example, the impact of \( \Delta \) 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\( \triangleright \text{logAll} \) to FilteredLog\( \triangleright \text{log} \).
- \( i_3 \): Introduction of a message send dependency from FilteredLog\( \triangleright \text{log} \) to Log\( \triangleright \text{log} \).

3.2 DELTA.ImpactFinder

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

We observe that the impact of a \( \Delta \) 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 \( \Delta \) introduces (or removes) into its origin branch is different from those introduced (or removed) when merging such \( \Delta \) in the destination branch.

![Figure 2. Impact of \( \Delta_F \) in A. In the origin branch, \( \Delta_F \) introduces three dependencies: one corresponding to an inheritance relationship, and the others corresponding to message sends.](image)

![Figure 3. Delta-impact of \( \Delta_F \) from A to B. Comparison of the impact of \( \Delta_F \) in both A and B. The dependency from Log\( \triangleright \text{logAll} \) to FilteredLog\( \triangleright \text{log} \) is missing in B.](image)
computes and shows the delta-impact of Section 2.

"Is the impact of \( \Delta \) in the current Pharo the same as the impact in the Pharo where \( \Delta \) 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 \( \text{Log} \), which the developer extends in his project by creating the subclass \( \text{FilteredLog} \). One year later, a new stable version of Pharo is available: version \( B \). Among plenty of changes in Pharo \( B \), the method \( \text{Log}\_\text{logAll} \) has been modified like in the FBCP example. As before, a bug appears in the method \( \text{logAll} \): when invoked in an instance of \( \text{FilteredLog} \). Note that when the developer loads the \( \text{FilteredLog} \) 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 \( \text{Log}\_\text{logAll} \) is the responsible. If a tool would have informed the developer that \( \text{Log}\_\text{logAll} \) changed, then he could save time.

We can pose this problem in terms of our approach. When the \( \text{FilteredLog} \) package is loaded in Pharo \( A \), some dependencies are introduced between code entities of \( \text{FilteredLog} \) 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, \textsc{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 \textsc{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.

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

To compute the delta-impact of \( \Delta \), we obtain the impact of \( \Delta \) 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 \( \Delta_F \) to the integrator would answer the questions Q1 and Q2 that we defined in Section 2.

\[
\begin{align*}
I(\Delta_F, A) : & \quad + \text{FilteredLog} \rightarrow \text{Log} \quad (i_1) \\
& \quad + \text{Log}\_\text{logAll} \rightarrow \text{FilteredLog}\_\text{log} : \quad (i_2) \\
& \quad + \text{FilteredLog}\_\text{log} : \rightarrow \text{Log}\_\text{log} : \quad (i_3)
\end{align*}
\]

\[
\begin{align*}
I(\Delta_F, B) : & \quad + \text{FilteredLog} \rightarrow \text{Log} \quad (i_1) \\
& \quad + \text{FilteredLog}\_\text{log} : \rightarrow \text{Log}\_\text{log} : \quad (i_3)
\end{align*}
\]

\[
DI(\Delta_F, A, B) = I(\Delta_F, A) \ominus I(\Delta_F, B) : \quad + \text{Log}\_\text{logAll} : \rightarrow \text{FilteredLog}\_\text{log} : \quad (i_2)
\]

\[(\text{Figure 4. DELTAIMPACTFINDER } I(\Delta_F, A) \) is the original impact of \( \Delta_F \), while \( I(\Delta_F, B) \) is the destination impact. \( DI(\Delta_F, A, B) \) is the delta-impact of \( \Delta_F \) 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 \textsc{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, \textsc{DeltaImpactFinder} can help Pharo integrators to discover semantic conflicts on the merge. Given a code change \( \Delta \) submitted by a developer, \textsc{DeltaImpactFinder} can answer:

\(^1\) 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-traits 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 → 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.

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.

Figure 5. Computation of the impact of $\Delta$ 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 $D(E + \Delta)$.
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. Ring code entities did not ensure isolation from tool’s environment, and they were not efficient to represent a whole Pharo environment.

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