Science of Computer Programming 96 (2014) 84–106

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Supporting streams of changes during branch integration

Verónica Uquillas Gómez [a,b], Stéphane Ducasse [a,b,*], Andy Kellens [a,b]

[a] *Software Languages Lab, Vrije Universiteit Brussel, Belgium*
[b] *RMoD, Inria Lille – Nord Europe and Université de Lille, France*

ARTICLE INFO

Article history:
Received 12 November 2012
Received in revised form 11 July 2014
Accepted 15 July 2014
Available online 30 August 2014

Keywords:
Branch
Source code changes
Stream of changes
Change dependencies
Merge

ABSTRACT

When developing large applications, integrators face the problem of integrating changes between branches or forks. While version control systems provide support for merging changes, this support is mostly text-based, and does not take the program entities into account. Furthermore, there exists no support for assessing which other changes a particular change depends on have to be integrated. Consequently, integrators are left to perform a manual and tedious comparison of the changes within the sequence of their branch and to successfully integrate them.

In this paper, we present an approach that analyzes changes within a sequence of changes (stream of changes): such analysis identifies and characterizes dependencies between the changes. The approach identifies changes as autonomous, only used by others, only using other changes, or both. Such a characterization aims at easing the integrator's work. In addition, the approach supports important queries that an integrator otherwise has to perform manually. We applied the approach to a stream of changes representing 5 years of development work on an open-source project and report our experiences.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

Version control systems (VCS) such as SVN, CVS and Git have become an indispensable tool for enabling teams of software developers to work together on a shared or distributed code base. Next to providing facilities for managing the source code of a system and maintaining that source code's history, these version control systems allow developers to work in *separate branches* of the system that later can be merged into the mainline of the system. Git, which is becoming increasingly popular, has placed branching at the center of its architecture and philosophy.

However, the task of understanding the consequence of a merge remains mostly manual and tedious due to a lack of practically applicable advanced tools. First, merging techniques used by popular VCS (*e.g.*, CVS, Subversion, Git) are based on simple, text-based algorithms, that only solve conflicts based on textual similarity, and are therefore *oblivious to the program entities* they merge. Even though there exist other approaches providing advanced merging support [2,17] that significantly reduce the amount of merging conflicts, such approaches do not support integrators in identifying redundant changes or changes that introduce inconsistencies at the level of the design of the target system. Second, there are no analyses to understand the dependencies between changes. The integrators are left to manually compare changes within the input stream of changes, and assess how these changes may impact the target system. Such work is particularly tedious between product forks, where the distance between branches grows larger over time.

In this paper we introduce a novel technique that tackles the above problems by modeling changes and the dependencies between them. Our technique, named *JET*, provides a first-class representation – based on the information contained in

* Corresponding author.

http://dx.doi.org/10.1016/j.scico.2014.07.012
0167-6423/© 2014 Elsevier B.V. All rights reserved.

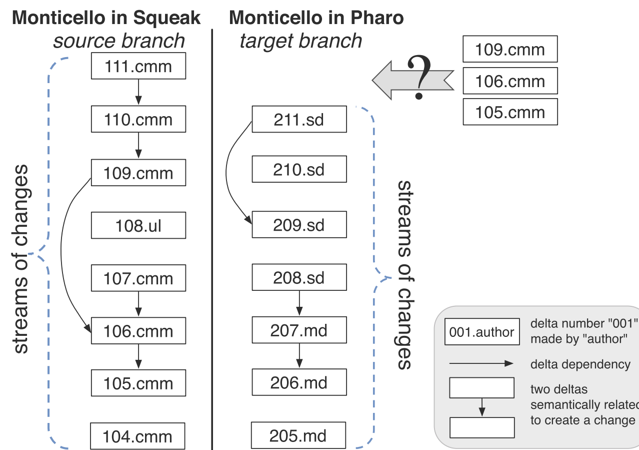

Fig. 1. Two branches of the Monticello versioning system and their stream of changes.

a version control system – of the history of the source code of a given system. By explicitly representing the changes between versions, and the dependencies between these changes, we provide additional information that guides integrators during the integration of changes. Such information is accessible for the integrators by means of simple queries (changes a certain change relies on, callers of a changed method) complemented by a dedicated dashboard and visualization that aid in comprehending sets of changes and the dependencies between such changes. We provide an implementation of our approach in Pharo.[1]

To illustrate our approach, we apply it to a concrete case study: the Squeak[2] forked versions of Monticello (a versioning system). We show how our approach aids in integrating forked versions of Monticello into the main distribution of Pharo. After forking, various components – such as Monticello – have evolved independently within Pharo and Squeak. We show how our tools aid in (a) cherry picking changes from this open-source project, (b) assessing the scale and impact of the changes, (c) determining which other changes these changes depend on, and (d) filtering irrelevant changes.

The contributions of this paper are: (1) A change and dependency model, as well as the algorithms for supporting streams of changes analyses. (2) A tool that provides lists of changes, deltas and dependencies of a stream of changes, along with a visual map of dependencies, and a browser to explore the history of any change within a given branch taking the dependencies between changes into account. (3) A qualitative assessment, in the context of a real-life open-source system, of our approach and tools.

2. Challenges in supporting merge operations

While merging tools support automatic merging of *textual* modifications to text files, the real challenge lies in taking into account the actual *contents* of the modifications during the merging process. The following example from Pharo/Squeak illustrates the problems faced daily by integrators that need to merge features in presence of change dependencies (by dependencies we mean that a change requires another one to achieve its purpose).

2.1. Task examples

Fig. 1 shows two streams (sequences) of changes in both branches of the Monticello core package. The integrator working on the target branch would like to understand the changes that have been performed in the source branch so that he can integrate some of the changes into the target branch.

Each node represents a *delta* (*i.e.*, a set of changes extracted from two versions). Note that there can exist dependencies between these deltas (indicated as directed edges), and that the numbers of the deltas in the *source* branch are unrelated to the numbers of the deltas in the *target* branch.

With current-day tool support, the integrator has to navigate the source branch *manually* to recover such dependencies between changes. Moreover, some part of the changes may conflict with the current target branch. Again these have to be identified manually. For a deep analysis of integrator tasks and needs refer to Chapter 3 of [22] which presents a full survey of integrator needs and questions. Several important tasks are summarized here as well.

Recover dependencies. The integrator has to navigate the source branch manually to recover the dependencies between the changes. As an example, consider the case in which an integrator wants to introduce the changes of the delta

[1] Pharo: http://www.pharo-project.org.
[2] Squeak: http://www.squeak.org.

109.cmm into the target branch. To do so, he has to check all previous changes to find out that delta *109.cmm* depends on delta *106.cmm*, which in turns depends on delta *105.cmm*. Therefore these three deltas will probably need to be integrated together. To discover such dependencies, he has to check out each individual commit, read the code and perform some analysis (taking notes about entities and their relationships) to extract the dependencies. Doing such task manually is a daunting task.

Assessing impact. Other problems left to the integrator are assessing the impact of integrating these changes into the target branch, and determining how these changes can be integrated without breaking the system or without introducing unwanted features.

Understand changes. Some part of the changes may conflict with existing features/implementations of the current target branch. Here the term conflict does not refer to a simple textual conflict that arises when two parts are edited concurrently. We refer to a conflict of features when one change does not raise a textual conflict but may lead to a different program behavior. Again the integrator has to identify such problems manually. He should check out the code and compare it with the existing one. He also has to understand how the changes would get invoked by the existing code and what is the impact of the changes on the existing applications.

Navigate sequence of changes. The integrator may want to know if a given method has been changed multiple times with a branch. Knowing such changes may help him to minimize his work. He may also want to know if a new method has been used afterward by future changes with a branch. Getting such information is another task that is overly time consuming without tool support.

2.2. Current solutions and limits

Some approaches exist to support the integrator in his tasks but they are limited.

Textual merge. While version control systems offer support for merging versions, this is mostly limited to a textual merge. Such systems do not take into account semantics[3] of the (object-oriented) programming language used or how the merged changes potentially introduce conflicts. Even a system such as Darcs, with an advanced model of changes allowing change permutations, does not take into account the language semantics for the merge resolution. In these cases, it is up to the integrator to analyze the changes manually and assess whether it is feasible to merge these changes, how they impact the branch and how they can be integrated.

Cherry picking. The task of merging non-trivial changes between various branches of a software system is still done largely manually. Especially in the case where the branches to be merged have evolved independently and therefore drifted apart, and automatic merging leads to an abundance of conflicts, or where an integrator wants to integrate code changes from one branch into another (known as *cherry picking*). Merging these changes can be tedious and time consuming.

Over time, it becomes increasingly difficult for a branch integrator to determine whether a change from another branch is relevant, whether the resources the change requires are available in the integrator's branch, whether the change will break the invariants of his branch, and how the change relates to any customizations he may have introduced.

Commit and branch history. Modern tools offer dedicated UI showing the branch and merge of projects. However, they work at a textual level and ignore the model of the subject that they are versioning. The integrator gets some help navigating the commits and branches but this is insufficient.

Simple diffing. Simple diff tools like the one available on Github, Eclipse or the Pharo merge tools show the difference between pieces of text. Version control systems do not provide integrators with information about dependencies between changes (*i.e.*, which code is needed by a particular change to be semantically correct). Assessing which other changes are needed by a particular change has therefore to be done manually.

Our approach in a nutshell Our approach characterizes and contextualizes a change within the complete sequence of changes (stream of changes). By characterization, we mean the nature of the changes (addition, removal, modification), their location, their size, and their author. By contextualization, we mean whether a change is isolated, part of a large sequence of changes, and other information about the situation of a change within a group of changes. To this end, our approach provides a first-class representation of changes, along with support for calculating the dependencies between these changes. We complement this representation with a dashboard that supports the characterization of the changes and provides facilities for navigating and inspecting changes based on their dependencies.

3. JET: manipulating streams of changes

Fig. 2 shows the architecture of our approach. To extract changes made to a system, we analyze the information stored in VCS. Based on the *commits*, we construct the history of a system as a set of *snapshots*, each of which represents the complete

[3] By semantics we mean for example the meaning of visibility modifiers: in Java just changing a modifier (*e.g.*, a protected method into a private one) can alter the behavior of a program by breaking hook invocation.

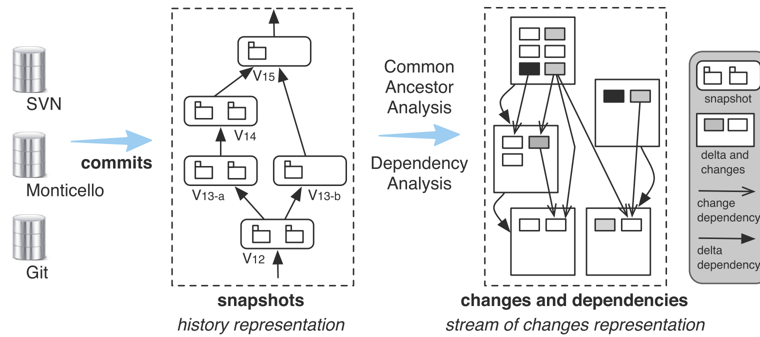

Fig. 2. JET architectural overview.

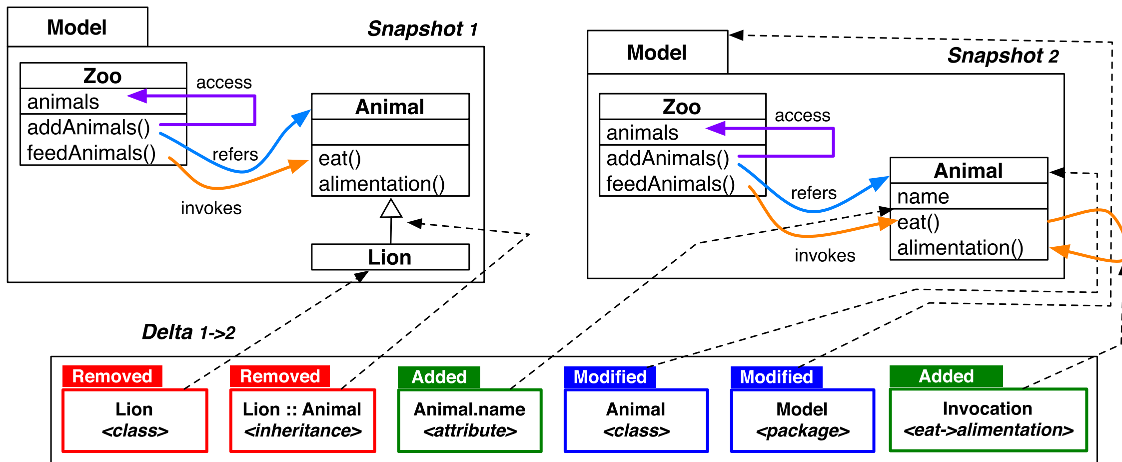

Fig. 3. A change is any alteration done to a program entity or relationship. An entity can be added, removed or modified.

state of the system at a given point in time. From these snapshots, we compute *deltas* (*i.e.*, a first-class representation of the *changes* between each snapshot and its predecessors), and *dependencies* between these changes and deltas. Finally, we characterize deltas and dependencies within the stream of changes.

We start our discourse by introducing the basic concepts and terminology used by JET. Next, we present how deltas and dependencies are calculated. We finish this section by showing the characterization of deltas and dependencies within the stream of changes.

3.1. Changes

Before introducing basic definitions we define the term *changes*: a *change* is any alteration done to a program entity or relationship. Fig. 4 shows the different entities and relationships we take into account. Since we manipulate different version of such entities, they are specialized into their history counterpart as shown in Figs. 5 and 6. Within JET, we represent each change as a separate entity. We distinguish between three kinds of changes, namely removals, additions or modifications, as shown in Fig. 3 and detailed later in Fig. 10.

Fig. 3 presents two snapshots of a simple system consisting of classes Zoo, Animal and Lion within the package Model. Associations are also represented: the method Zoo»addAnimals *refers* to the class Animal, the method Zoo»feedAnimals *invokes* the method Animal»eats, Lion *inherits* from Animal and the method addAnimals *accesses* the attribute animals. In the second snapshot the attribute name was added to the class Animal and the class Lion was removed. At the bottom, we see the six changes between these two snapshots: the changes are annotations of any of the entities and relationships present in the snapshots. In this case we have a *class modification* (Animal), a *package modification* (Model), an *attribute addition* (name), a *class removal* (Lion), a *method invocation addition* (eat() invokes alimentation()), and an *inheritance removal* (between Lion and Animal).

In addition, when we say that a change depends on another one, we mean that an entity from one delta has as target one entity of another delta. For the example above: the change (in $\text{Delta}_{1\rightarrow 2}$) representing the addition of the attribute name to the class Animal depends on another change representing the actual addition of the class Animal.

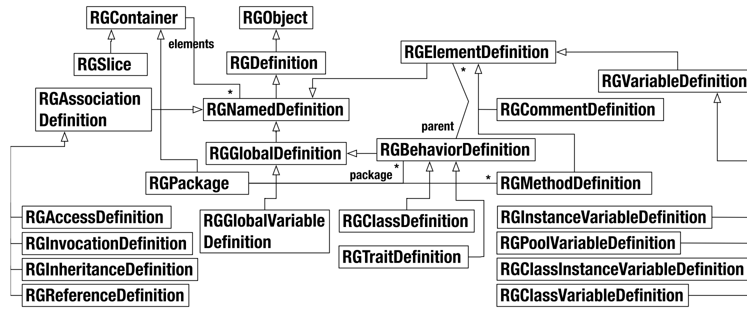

Fig. 4. Ring code meta-model with associations.

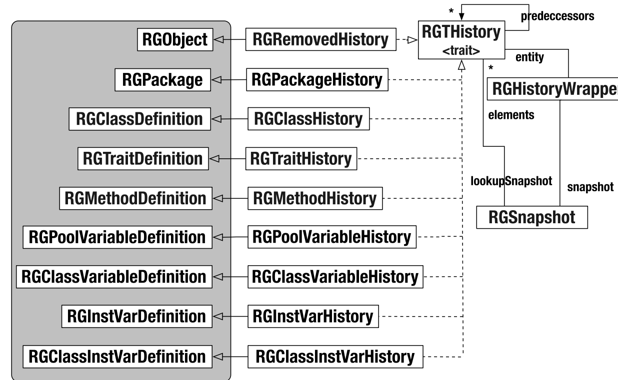

Fig. 5. HRing history meta-model – key program entities.

3.2. *Definitions for sequence of change modeling*

Now we define the terms and concepts that form the basis of our approach.

Program entities and relationships Our approach provides a representation of the program entities that are present in the history of a system, as well as the relationships between these entities. We model program entities and relationships as first-class objects using a *history* meta-model we built on top of Ring [24].

Ring: A source code meta-model. The Ring model contains a first-class representation of the packages, classes, traits, methods and attributes of a system; as relationships we consider attribute accesses, method calls, class references and class inheritance relationships. Fig. 4 shows the main elements of our model – similar to FAMIX [8] but adapted to represent Smalltalk source code: it models structural information packages (RGPackage), classes (RGClassDefinition), traits (RGTraitDefinition), methods (RGMethodDefinition) and attributes (RGInstanceVariableDefinition, RGPoolVariableDefinition, RGClassVariableDefinition and RGClassInstanceVariableDefinition). Then to support our dependency analysis four types of relationships are represented: access i.e., a method accesses an attribute (RGAccessDefinition), invocation i.e., a method invokes a group of potential other methods (RGInvocationDefinition), reference i.e., a method or class makes an explicit reference to another class either directly or via self/super (RGReferenceDefinition) and inheritance i.e., a class inherits from another one (RGInheritanceDefinition).

HRing: A history meta-model. Based on the Ring code meta-model, we define a meta-model taking into account the fact that we can have multiple versions of the same entity. Figs. 5 and 6 show the key classes of the history meta-model (shown without background) which extends the Ring meta-model (shown with gray background). The first figure illustrates the classes that model the history of program entities, and the second figure presents the classes that model the history of relationships between program entities. RGHistoryWrapper supports optimized navigation and queries between versions (an explanation of this concept lies outside the scope of this paper. For more information we refer to [15]).

Commits Developers publish source code modifications to a repository in the form of commits resulting in new revisions (also known as versions). A *commit* refers to the group of additions, modifications and removals made to program entities of a software system (as illustrated below).

Snapshots A *snapshot* is a set of program entities and relationships at a given point in time in the history of a system. This set of entities represents the complete system under analysis, in contrast to commits that refer to the changes submitted

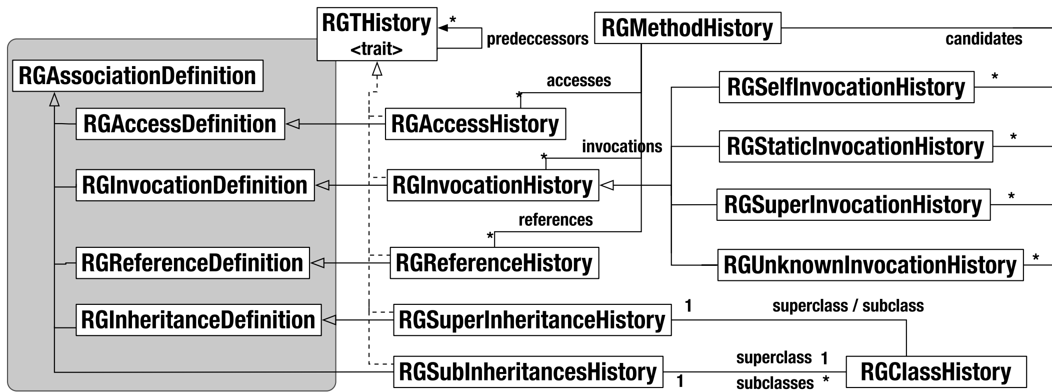

Fig. 6. HRing history meta-model – relationships.

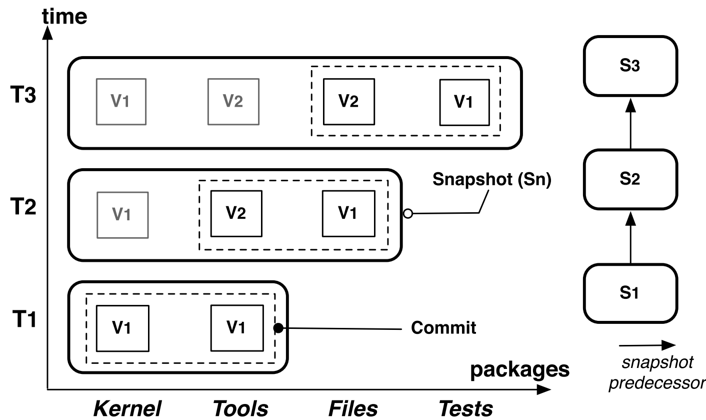

Fig. 7. Example of how a snapshot is determined.

at a point in time. A snapshot is derived from a commit. However, it also includes unchanged program entities and relationships present in the history at that point in time. We build snapshots of a system by analyzing the commits contained within a Monticello repository.[4]

Fig. 7 illustrates this definition of *snapshots*. Along the x axis we see the different packages that are contained within a repository. The y axis represents the various points in time at which a commit occurred. At time T_1 the first version V_1 of packages *Kernel* and *Tools* were published. Both versions represent the first commit and also the first snapshot S_1. Next, at T_2 a new version was committed of the previously existing package *Tools* (V_2), and a newly created package *Files* (V_1) was added to the system. The changes made to these two packages correspond to the second commit and hence also snapshot S_2. Note that the second snapshot S_2 also includes the package *Kernel* (V_1) that was already present in the repository. Finally, at T_3 a third commit was published containing a new version of the package *Files* (V_2) and the first version of the package *Tests* (V_1). The third snapshot S_3 includes both packages and also includes the unchanged packages *Kernel* (V_1) and *Tools* (V_2) as they were part of the system at that time.

Deltas A *delta* is a set of changes representing the differences between two successive *commits* or *snapshots* (known as snapshots S_{base} and S_{target}) present in the history. We illustrated a delta in the running example shown in Fig. 3. Within JET, we provide a first-class representation of each delta that keeps track of its predecessor(s) and successor(s), which allows us to create a graph of deltas. Fig. 8 shows a graph of snapshots (rounded rectangular shapes) and the deltas (rectangular shapes) extracted from these snapshots. Snapshots and deltas are linked to their predecessors. For example, the delta $D_{1\to2}$ represents the differences between the snapshots S_1 and S_2.

Note that our snapshot graph can contain merges. For example, the last snapshot S_5 is the result of merging S_4 and S_3. In such cases, our graph of deltas will contain a delta for each predecessor of the merged snapshot. This results in delta $D_{4\to5}$ from S_4 to S_5 and delta $D_{3\to5}$ from S_3 to S_5.

[4] In Monticello, the versioning system supported by our tools, each package is versioned individually. To determine which versions of the packages belong together (*i.e.*, represent a commit), we use a sliding window technique [25] that considers that multiple packages belong to the same commit if they are committed by the same author within a time interval of 5 minutes.

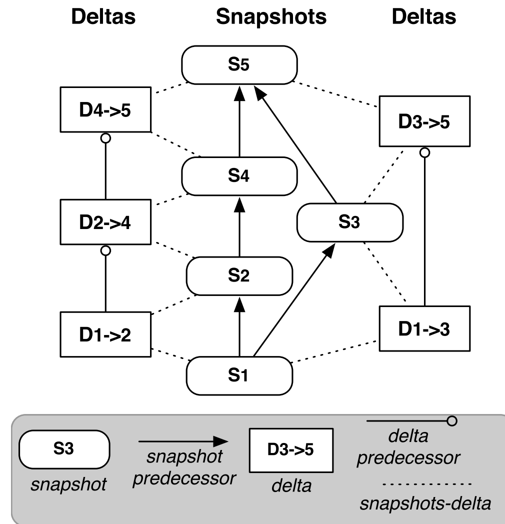

Fig. 8. Snapshots and deltas.

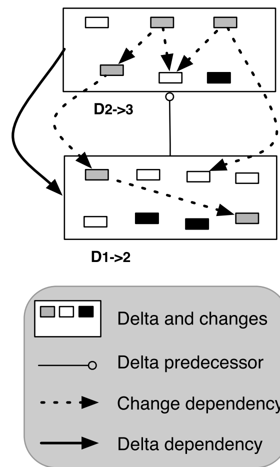

Fig. 9. Change and delta dependencies.

Change dependencies A *change dependency* captures the fact that a given change CH_y potentially depends on another change CH_x (*i.e.*, $CH_y \rightarrow CH_x$). For example, if a modification to method M_{foo} adds a call to a new method M_{bar}, this change introduces a *change dependency* of M_{foo} to M_{bar}. That means that in order to integrate the modified method M_{foo}, the added method M_{bar} is needed. Such a dependency can exist between changes within the same delta or between changes in different deltas.

Delta dependencies A *delta dependency* expresses a dependency from delta D_n to delta D_m (*i.e.*, $D_n \rightarrow D_m$), where a change CH_y in D_n depends on a change CH_x in D_m (*i.e.*, the change dependency $CH_y \rightarrow CH_x$ exists). That means that a delta depends on another delta if any change within it depends on a change in other delta. Considering the example presented in the definition of change dependencies but assuming that the method M_{bar} was added in delta D_7 and that the method M_{foo} was modified in delta D_8, then due to the change dependency between both changes, the *delta dependency* $D_8 \rightarrow D_7$ is introduced.

Fig. 9 shows *change dependencies* using directed dashed lines and *delta dependencies* using directed lines. The delta $D_{1\rightarrow 2}$ and its successor delta $D_{2\rightarrow 3}$. Delta $D_{2\rightarrow 3}$ contains 2 changes that depend on 2 other changes of the same delta, and 2 changes that depend on changes contained in the predecessor delta $D_{1\rightarrow 2}$. Therefore the delta $D_{2\rightarrow 3}$ has a *delta dependency* on its predecessor $D_{1\rightarrow 2}$.

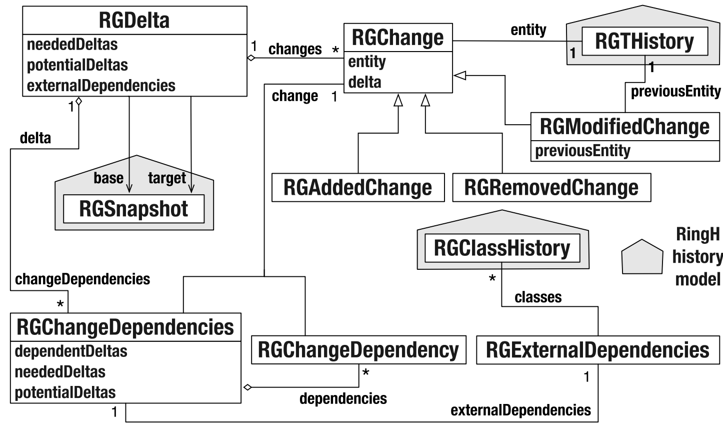

Fig. 10. Change and delta meta-model – key classes.

3.3. Change and dependency model

After defining the concepts that are used in JET, we discuss in more detail how we model changes and dependencies between changes, and how such a model is built using the history model described above. An overview of the main entities of the change and delta model are shown in Fig. 10:

- RGChange is the root class that models changes. A change wraps the element (entity) that changed in the stream, creating a link to the history model (shown with a gray background). For each kind of change, we provide a separate subclass, namely RGAddedChange, RGRemovedChange and RGModifiedChange. A RGModifiedChange keeps track of the previous state of the modified entity (previousEntity). Finally, a change also knows from which delta it originated.
- RGChangeDependency is the representation of a dependency between two changes. This dependency can exist between changes within the same delta or between changes in different deltas. Our approach characterizes a change dependency based on the locality and size of potential changes that can satisfy this dependency (as explained in Section 4).
- RGExternalDependencies models a dependency between a change and program entities that are not present in the stream of changes. Such external classes are represented as stub classes in the history model.
- RGDelta represents the differences between two snapshots in terms of a set of changes. It contains a set of RGChange objects and – indirectly – the associated set of RGChangeDependency objects. From the set of change dependencies, a delta is able to characterize its dependencies (as explained in Section 4).

3.4. Delta dependency mechanism

In what follows we describe our algorithm to calculate the dependencies between deltas. Our algorithm takes as input a set of snapshots and computes a change-based representation of this set of snapshots, along with the dependencies between and inside deltas.

We divide our algorithm in two different stages: 1) calculating deltas, and 2) finding dependencies. For each stage, we introduce its algorithm and then we proceed to explain the main steps of the process.

Stage 1: Calculating deltas

1. Find all root snapshots (snapshots with no predecessors)
2. For each root snapshot:
   (a) **Calculate a root delta** containing additions of the snapshot's elements
   (b) Traverse the graph of snapshots from the root to its most recent successor(s)
      - **Calculate a delta** for each pair (predecessor, successor) containing their differences
3. Assemble a graph of deltas by finding the predecessors of each delta
4. Retrieve all merged snapshots (snapshots with more than one predecessor)
5. For each merged snapshot, **refine related deltas** by taking their common ancestor into account

Calculating root deltas We consider all snapshots that do not have predecessors to be root snapshots. Earlier, we have defined a delta as the representation of the differences between two snapshots. As *root snapshots* do not have any predecessors, we introduce here the notion of *root deltas*. We compute a root delta by creating RGAddedChange objects for each of the program entities and relationships defined in the root snapshot.

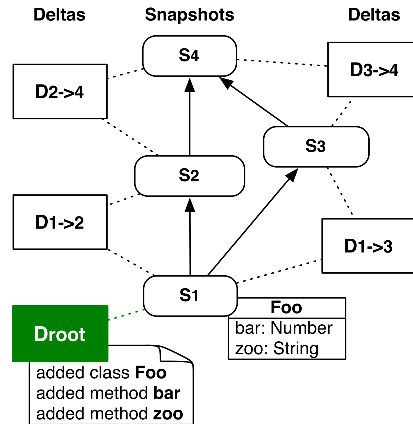

Fig. 11. Root delta: completing a change-based representation of a stream of changes.

Fig. 11 shows a graph of snapshots (in the middle) containing one root snapshot S_1. The root snapshot contains a single class Foo and two methods bar and zoo. Therefore, the *root delta* D_{root} (shown in green) contains *added changes* for each of these program entities: an added class for Foo, and two added methods for bar and zoo. Note that from each method's body, other *added changes* may be created to represent method calls, class references and attribute access. We omit the relationships here to avoid cluttering the figure.

Calculating deltas A delta is computed by extracting the differences between a pair of snapshots (predecessor, successor). The differences are then reified as changes and represented as additions, modifications and removals using the model presented in Fig. 10.

We consider that an entity has been modified when its definition has been changed:

- *Package.* A new subpackage was added or an existing subpackage was removed.
- *Class.* Its definition or its comment changed.
- *Method.* Its source code or its protocol changed.

The deltas are calculated by traversing the graph of snapshots, starting with the root snapshots until we reach snapshots that do not have any successors.

We illustrate this process in Fig. 11. The root snapshot S_1 has two successors S_2 and S_3. For each pair (predecessor, successor) a delta is calculated. When applied to our example, this results in two deltas: delta $D_{1\to 2}$ for the pair (S_1, S_2) and delta $D_{1\to 3}$ for the pair (S_1, S_3). After processing snapshot S_1, we continue traversing the graph via the successors of S_2 and S_3. The snapshot S_2 has a successor S_4 which results in the delta $D_{2\to 4}$. Finally, the snapshot S_3 has a successor S_4 which results on the delta $D_{3\to 4}$.

After traversing the whole graph of snapshots and computing deltas, we assign the predecessors and successors of each delta based on the predecessors and successors of the snapshots that generated such delta. The root delta D_{root} has two successors $D_{1\to 2}$ and $D_{1\to 3}$. $D_{1\to 2}$ has as predecessor D_{root} and as successor $D_{2\to 4}$. Finally, $D_{1\to 3}$ has as predecessor D_{root} and as successor $D_{3\to 4}$.

Refining deltas in the presence of merge Up until now, our calculation of deltas does not take into account that a snapshot might be the result of merging two snapshots. In Fig. 11, the snapshot S_4 is a merge between the snapshots S_2 and S_3. For each of these predecessors, we have calculated a separate delta. However, due to the merge, each of these deltas might be "polluted" with a number of changes that might have occurred in the other branch. As we are only interested in the changes that *contribute* to a merged snapshot, we perform a post-processing step on all deltas that are associated with a merge. To this end, we propose a technique similar to three-way merging algorithms [16].

We illustrate our technique by means of a slightly more complex scenario, as shown in Fig. 12. In this scenario, we have a snapshot S_{10} that is the result of merging S_3 and S_9. Note that both predecessors have a common ancestor – namely S_2. Assuming that S_3 is an older snapshot than S_9, the delta $D_{3\to 10}$ (shown in orange) potentially contains a number of changes that have occurred somewhere in the snapshots in between of S_4 and S_9, but that are unrelated to the changes of S_3 that contribute to S_{10}.

In other words, we are interested in all the changes that have occurred between snapshots S_2 and S_3 (indicated with the blue dashed lines) together with all the changes between S_3 and S_{10}, minus the changes happening in the other branch from S_2 to S_{10}. If we generalize this, we obtain:

$$D_{op\to m} - D_{ca\to m} + D_{ca\to p}$$

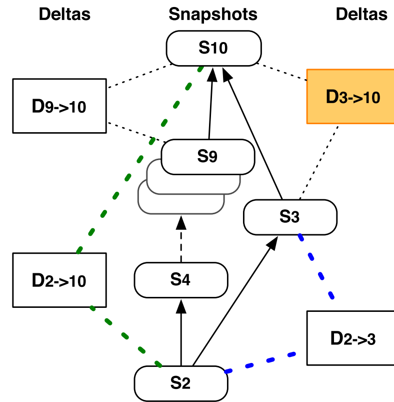

Fig. 12. Deltas in the presence of merge: taking into account common ancestor S_2.

where *op* is the oldest predecessor, *m* the merge, *ca* the common ancestor, and *p* the predecessor.

Applying this formula to refine $D_{3\rightarrow 10}$, we see that the delta is obtained by computing $D_{3\rightarrow 10}$ (original delta) $-$ $D_{2\rightarrow 10}$ (indicated with the green dashed lines) $+$ $D_{2\rightarrow 3}$ (indicated with the blue dashed lines).

Stage 2: Finding dependencies In the second stage of our algorithm, we calculate the dependencies between deltas. For each delta:

1. **Filter the changes within the delta**: Only select those that *may* depend on another change:
   (a) Include additions and modifications of classes and methods.
   (b) Exclude modifications of methods that do not introduce new method calls or class references.
2. For each change that may depend on other changes, **determine its dependencies**:
   - A change to a class depends on:
     - The most recent change to its superclass.
     - If such a change does not exist, its superclass is an external reference. Add a dependency to this external reference.
   - A change to a method depends on:
     - The most recent changes to the potential receivers (*i.e.*, methods) of its method calls, and on the most recent changes to its class references.
     - If changes to a class reference do not exist, the method refers to an external class. Add a dependency to this external reference.
3. Prune **redundant delta dependencies**.

Filtering changes within a delta Not all changes within a delta can lead to the introduction of dependencies. As a pre-filtering step, we partition the changes within a delta into two groups: 1) changes that potentially depend on other changes, and 2) changes that do not depend on another change.

For the first group, we only consider additions and modifications of classes and methods that can result in the introduction of dependencies. The reason is that we only require dependencies that are needed when integrating changes, and therefore we do not include removals. Furthermore, we exclude modifications to methods that changed their source code without altering the set of method calls and class references. Changes to classes that did not change the superclass of the class are also filtered out. All other changes within the delta are considered to belong to the second group.

Determining dependencies We proceed to determine the dependencies for each change within a delta that was categorized as a change that may potentially depend on other changes. A *change dependency* is a relation between two changes, where both changes can be present in the same delta or in different deltas. Changes to classes and methods can depend on other changes based on the following rules:

- **Class level**: Changing the superclass of a class introduces a dependency on the most recent change to the class' superclass.
- **Method level**: Changes to a method depend on:
  - **Change to class references**: The most recent changes to the referred classes.
  - **Change to method calls**: The most recent changes to the potentially called methods (*i.e.*, candidate set[5]).

[5] The set of potentially called methods is approximated statically taking into account polymorphism.

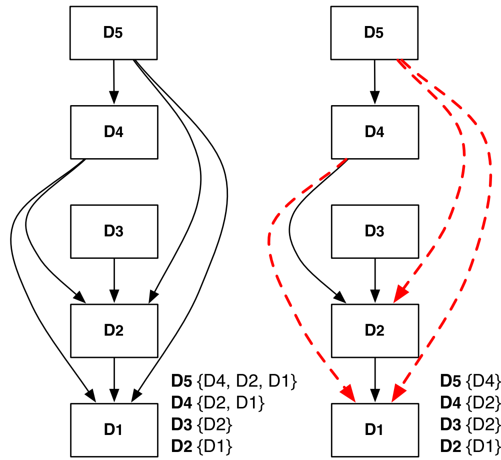

Fig. 13. Redundant delta dependencies.

Type of delta	Is dependent	Is a dependency
Source		x
Intermediate	x	x
Island		
End	x	

Fig. 14. Types of deltas by the presence of dependencies.

To determine the most recent change of an entity, we make use of the graph of deltas that was determined in a previous step of our algorithm. Based on the dependencies between changes, we also compute the dependencies between deltas (*delta dependencies*). We say that delta D_1 is dependent on another delta D_2, if there exists at least one change in D_1 that depends on a change in D_2.

Pruning redundant delta dependencies Note that our algorithm for calculating delta dependencies can result in redundancies. To illustrate this, consider the left graph of deltas depicted in Fig. 13; delta dependencies are indicated by means of a black directed edge. If we take a look at delta D_4, we see that it depends on deltas D_2 and D_1. However, since delta D_2 also depends on delta D_1, the dependency between D_4 and D_1 is redundant as it is already implied by the configuration of deltas. Likewise, delta D_5 depends on three deltas (D_4, D_2 and D_1) of which the dependencies $D_5 \rightarrow D_2$ and $D_5 \rightarrow D_1$ are also implied by the chain of dependencies $D_4 \rightarrow D_2$, and $D_2 \rightarrow D_1$. Therefore, these redundant delta dependencies (indicated by means of a red directed dashed edge in the right graph of deltas) can be safely pruned.

4. Characterizing deltas and dependencies within the stream

Based on our dependency analysis between deltas, we provide a characterization of these deltas within a stream of changes. The goal of this characterization is to speed up the process of understanding a change, its context and its dependencies, and support integrators in the decision-making process regarding the integration of changes. For example, this information can aid integrators in filtering changes that are irrelevant in a particular context and that should not be integrated, in prioritizing which changes to integrate first, and so on.

4.1. Presence of dependencies

As a first criterion for characterizing dependencies we consider whether a delta has dependencies or not, and the directionality of these dependencies. As mentioned earlier, a delta can depend on other deltas, and a delta can be the dependency of other deltas.

We classify deltas depending on the existence of such dependencies. This classification provides an initial indication of the complexity of a delta and is therefore potentially valuable to an integrator. In Fig. 14 we present the four types of deltas along with an illustrative example shown in Fig. 15.

- **Island**: a delta that does not depend on another delta and is not the dependency of any delta. *Islands* are the simplest type of delta; integrating them only requires the changes in the delta to be processed.
- **Source**: a delta that has no dependencies but is a dependency of other deltas. *Sources* can still be considered as simple cases as no other changes need to be analyzed beforehand.

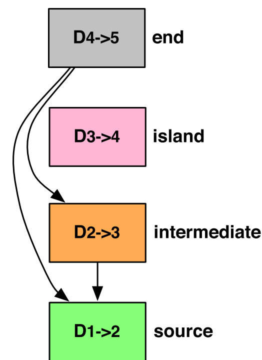

Fig. 15. $D_{3\rightarrow 4}$ is an *island*, $D_{1\rightarrow 2}$ is a *source*, $D_{4\rightarrow 5}$ is an *end*, and $D_{2\rightarrow 3}$ is an *intermediate*.

- **End**: a delta that depends on other deltas but no other delta depends on it. An *end* is already a complex delta, because it has to be integrated together with the deltas it depends on.
- **Intermediate**: a delta that depends on, and is the dependency of other deltas. They are the most complex deltas and the ones that should be integrated carefully.

4.2. Type and cardinality of change dependencies

The changes belonging to a single delta can require the presence of certain source code entities that were introduced or changed in preceding deltas. As a second criterion, we distinguish between dependencies that can or cannot be found within the stream.

- **Local**: a dependency is local when the entities it depends on exist within the stream of changes.
- **External**: a dependency is external when the entities it depends on do not exist within the stream (*e.g.*, method printOn: refers to class Set, but Set is a library class that was not introduced in the stream of changes).

As we are analyzing object-oriented programming languages, this introduces a level of uncertainty (*e.g.*, in the case of polymorphic calls). As a third criterion, we consider for a particular *change dependency* whether multiple candidates may be present within the stream.

- **Unique**: a change dependency is unique if there is only one potential candidate in the stream (*e.g.*, method foo calls method bar; there is only one implementor[6] of bar in the stream).
- **Multiple**: a change dependency is multiple if for a single dependency there are multiple potential candidate changes in the stream on which it depends (*i.e.*, due to polymorphism, lack of type information, and so on. *e.g.*, method foo calls method bar; there are four implementors of bar in the stream).

Delta dependency classification Based on our characterization of deltas, we also provide a characterization of the *delta dependencies*:

- **Needed**: a delta D_1 is a *needed delta dependency* for delta D_2 if at least one change in D_1 is the *unique change dependency* of a change in D_2. In other words, in order to integrate delta D_2, we need to also analyze the changes in D_1.
- **Potential**: a delta D_1 is a *potential delta dependency* for delta D_2 if there are changes in D_2 with *multiple change dependencies* and at least one of these change dependencies belongs to D_1. In other words, in order to integrate delta D_2, an integrator *will* need to analyze these change dependencies in D_1, to be safe.
- **External**: we say that a delta has *external dependencies* if at least one of its changes requires an entity that is not present within the stream.

The JET dashboard presents deltas and dependencies by using these characterizations. The JET map, however, only displays needed dependencies to simplify the view. Furthermore, which priority is given to the different types of deltas, or which priority is given to the different types of delta dependencies is up to the integrators.

[6] An implementor is a class or a trait defining a method here the method bar.

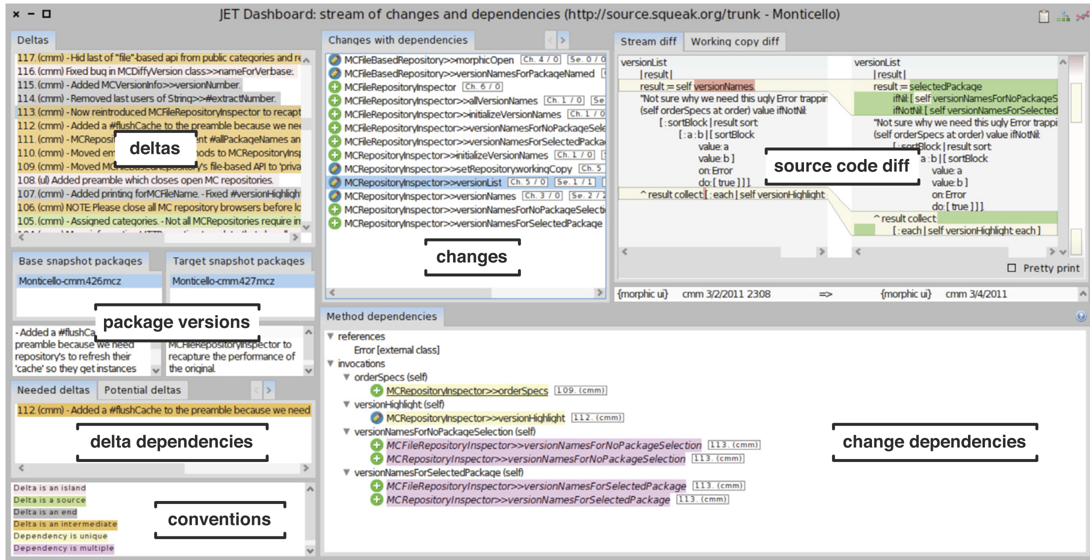

Fig. 16. The JET dashboard and its main elements.

5. Stream change analysis in early integration phase with JET

Our approach, *JET*,[7] provides tool support to integrators to characterize and understand streams of changes and their dependencies. The JET dashboard presents lists of deltas, lists of changes per delta, lists of dependencies per change, and lists of dependencies per delta. Moreover, the dashboard adds several metrics to each change such as the number of times that an entity has been changed, the number of callers and implementors of a method in a single version, or throughout the stream of changes.

The textual information provided by the dashboard is also complemented by the JET map, a visualization displaying deltas with their dependencies. The map provides a visual display of a number of metrics such as the number of dependencies of a delta, the number of deltas that depend on a certain delta. Using a color convention, the map displays a characterization of a set of deltas following the criteria discussed in Section 4. Finally, JET provides several utilities to integrators that allow them to filter and navigate dependencies and deltas.

In its current state, JET is intended to be loaded into the Pharo system in which the integration of changes is done. This also allows us to access the current working copy (a.k.a image) so that an integrator not only can assess a stream of changes with respect to its history, but also with respect to the already integrated source code present in the image.

5.1. The JET dashboard

The structure and main elements of the JET dashboard are shown in Fig. 16. The dashboard offers textual information extracted from the stream of changes, such as deltas and dependencies, and also allows an integrator to access the whole history in detail. For the sake of brevity, we only present an overview of each element of the dashboard.

Deltas. Deltas are extracted from the snapshots following the principle explained in Section 3.4. The first panel of the dashboard lists all deltas, thereby maintaining the order of the snapshots. To aid integrators in finding the delta they are looking for, the label of each delta is composed of the commit number, the name of the author of the associated commit, along with a summary of the commit message. For example, delta *179.cmm – Fix for package renaming* corresponds to commit number 179, committed by *cmm* (*i.e.,* alias of the author), and it fixed a bug.

Package versions. Each delta represents the changes that happened between a base and a target snapshot. This panel lists the package versions that are included in both snapshots (*e.g., Monticello-cmm.426* corresponds to the version *426* of the package *Monticello* committed by *cmm*). For each package version the complete commit message is provided as well.

Changes. The changes to packages, classes and methods belonging to a delta are classified into two lists: *Changes with dependencies* and *Changes without dependencies*. Both lists allow an integrator to inspect all changes of a delta and their evolution within the stream.

[7] JET: www.squeaksource.com/JET.

For each change, this panel provides metrics about the *number of times that the entity changed* (*e.g.*, *Ch. 5 / 2* corresponds to *5* changes over the total stream and *2* more changes in two later deltas). For a change to a method m, we also show the *number of callers* and the *number of implementors* (*e.g.*, *Im. 3 / 5 / 4* corresponds to *3* classes implementing a method with the same name (selector) m, *5* classes implementing m in a later delta, and *4* classes implementing m in the working copy).

Source code diff. The source code of a change is shown in this panel named *Stream code* or *Stream diff*. Additions and removals show the plain source code that was added or removed, and modifications display a diff highlighting the part of the code that changed (in green or red for additions and removals respectively). Moreover, if the changed entity (*e.g.*, method) exists in the working copy, an extra diff (*Working copy diff*) will appear comparing both. By providing the *Working copy diff* an integrator not only can inspect the code that changed within the stream but can also compare that code to the current code of the system. Finally, additional information about the change is displayed, such as the author that changed the entity and the timestamp of the change.

Change dependencies. We provide a panel that presents the change dependencies of methods and classes grouped by invocations (method calls), class references and superclasses. Each dependency indicates the change associated to it and the delta to which that change belongs. For example, the *added* class MCFileRepositoryInspector of delta *113.cmm* depends on the superclass MCRepositoryInspector. Since this superclass was most recently modified in delta *112.cmm*, there exists a change dependency and therefore a delta dependency (*113.cmm* → *112.cmm*). From this panel, an integrator can also filter change dependencies, or inspect which of the changes of the delta are a dependency of another delta (*e.g.*, delta *124.cmm* depends on delta *113.cmm* for the *modified* method MCRepositoryInspector»refreshEmphasis (of *124.cmm*) that calls the *modified* method MCRepositoryInspector»identifyNewerVersionsOf: (of *113.cmm*)).

Delta dependencies. This panel presents the characterization of delta dependencies for a particular delta based on the three categories discussed in Section 15: *Needed dependencies*, *Potential dependencies*, and *External dependencies*. To ease navigation, these categories are complemented by a four list showing the *Deltas depending on me*. For example, the *end* delta *115.cmm* needs the *intermediate* delta *112.cmm* and potentially the *end* delta *85.bf*. It has one external dependency to class HTTPSocket, and there are no deltas depending on delta *115.cmm*.

Conventions. Colors are used to represent types of deltas and dependencies (introduced in Section 4). They help integrators get instantaneous information and reinforcement of their knowledge. The conventions are the same in the entire dashboard: pink for *island* deltas, green for *source* deltas, gray for *end* deltas, orange for *intermediate* deltas, yellow for *unique* change dependencies and magenta for *multiple* change dependencies. Font styles are used to complement dependencies, italic for change dependencies within the same delta, and underlining for *redundant* dependencies. Icons are also used to represent each kind of change (green plus for additions, blue pencil for modifications and red minus for removals).

5.2. The JET map

The JET map is a visualization that provides an overview of how deltas and dependencies are linked together. The map, as shown in Fig. 17, mainly offers a simplified view of the dashboard information and aims at guiding integrators in determining where to start the analysis of a stream of changes. In a sense, this visualization provides integrators with a means to assess how complex it is to integrate the changes of a particular delta.

The map visualizes *source*, *intermediate* and *end* deltas that have dependencies or serve as dependencies of other deltas. Therefore *island* deltas are omitted. Rectangles represent deltas and directed edges represent delta dependencies. A rectangle includes the label of a delta (number and author). The height of a rectangle (delta) is related to the number of deltas that depend on this delta. The border width of a rectangle is related to the number of dependencies of this delta. The color convention used in the dashboard is used in the map as well.

Fig. 18 takes a more in-depth view of two deltas. The example at the top displays the *intermediate* delta *112.cmm* (the orange node in the middle) that only depends on the *intermediate* delta *111.cmm*, therefore the border of *112.cmm* is thin. Red directed edges are used to point to the dependencies of a delta. Twelve deltas depend on *112.cmm* which makes the height of its rectangle considerably larger than the other visualized deltas. Blue directed edges indicate which deltas depend on a particular delta. The example at the bottom shows the *end* delta *159.fbs* (in gray) that has three dependencies on *source* deltas *143.kb* and *6.bf* (in green), and on *intermediate* delta *112.cmm* (in orange). Thus the border of the node in this case is thicker. As this is an *end* delta, meaning that no deltas depend on it, the height of the rectangle has the smallest possible value.

Finally, the map also offers textual information as a *fly-by-help*. For a delta it shows the commit messages; for a dependency it shows the deltas involved and their commit messages. It should be noted that we did not spend a large effort to work on adapted algorithms to place adequately the nodes and we consider that the visualization could be largely improved.

5.3. Querying a stream of changes with JET

JET complements the information provided by the dashboard and the map with a third browser, the *JET Query Browser*. The goal of this browser is to aid integrators in understanding the *complete* history of an entity, together with its depen-

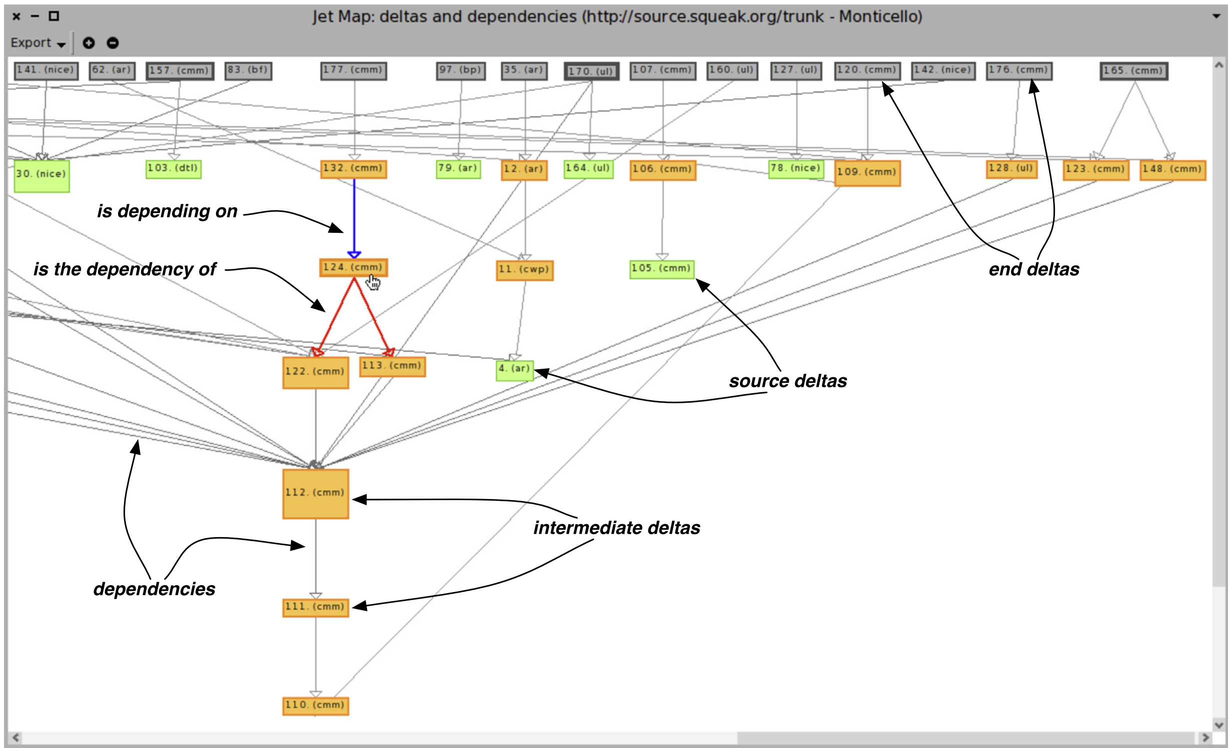

Fig. 17. The JET map: green nodes are *source* deltas (not depending on others), orange nodes are *intermediate* deltas (having dependencies and others depending on it) and gray nodes are *end* deltas (only depending on others). The island deltas (without dependencies) are not shown in the map. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

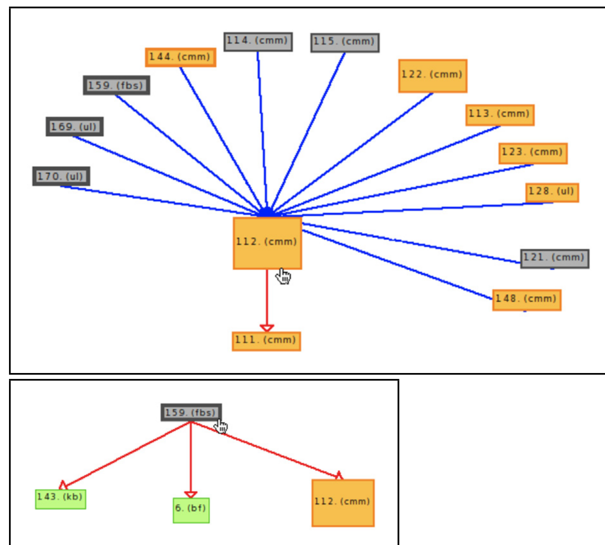

Fig. 18. Deltas and dependencies on the map.

dencies and users at any point in time (*i.e.*, within a delta). This information is vital for answering integrator questions[8] such as "Is this change still the most recent one?" ["Is there any later change in the sequence that supersedes it?"]. With this browser an integrator can know if a change introduced at one point is used by the following changes. Note that this

[8] The PhD [22] contains a list of questions that integrators ask when they perform a change integration.

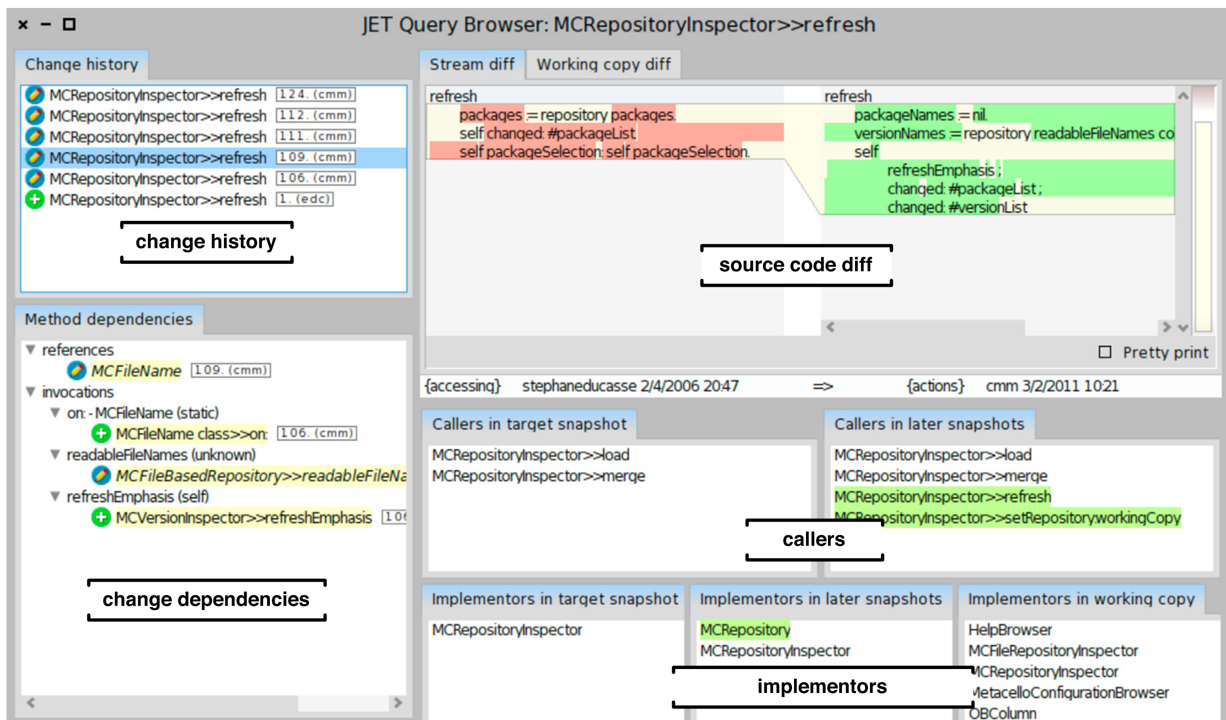

Fig. 19. The JET query browser and its elements.

browser complements the dashboard, which provides a set of metrics per change, with more fine-grained information about the changes.

The structure of the query browser is shown in Fig. 19. In the following we summarize each of the components of the browser.

Change history. The changes of an entity are listed in the first panel of the browser. This shows how an entity has evolved within the stream, and therefore already answers several questions related to the frequency of a change, who are the authors, where and when was that entity changed, which are the later changes. For each change, the kind of change is shown by means of an icon (addition, modification, removal) as well as the delta in which the change occurred.

Source code diff. This element of the browser is the same as described in Section 5.1. The advantage of providing this component is that for a particular entity the integrator is able to explore how the code evolved, together with who (author) changed the code and when (timestamp).

Change dependencies. The dependencies of a change are listed in this panel. This component is part of the dashboard as well, and it was explained in Section 5.1. While in the dashboard changes are classified per delta, the query browser allows comparing how the dependencies of a particular change evolved over time.

Callers. For a *method change* this element of the browser presents two lists. The callers in the first list are extracted locally, taking only the current delta into account (*i.e.*, the target snapshot of the delta containing such change). The callers in the second list take into account the subsequent changes in the stream from later points in the history (snapshots that are successors of the target snapshot). This shows whether a method is actually used within the delta and also how important it is for the subsequent deltas. Callers that are removed later in the stream appear with a light red background in the first list. Callers that are added later in stream appear with a light green background in the second list.

Implementors. This element of the browser presents three lists. As for callers, the two first lists present the classes implementing a method with the same selector of the *method change* extracted from the current delta and from subsequent deltas. Both lists follow the same coloring convention. The third list corresponds to the classes implementing that selector in the working copy. An integrator can compare the classes that implement a selector within the stream of changes to the current classes of the system in Pharo that implement the same selector.

Table 1
Size of Monticello.

Description	Squeak	Pharo
Class	117	116
Method	1559	1587
LOC	7739	8083

Table 2
Summary of changes and deltas.

Description	#
Changes *with* dependencies	1909
Changes *without* dependencies	1639
Total changes	*3548*
Intermediate deltas (orange)	18
Island deltas (pink)	105
End deltas (gray)	49
Source deltas (green)	22
Total deltas	*193*

6. Integrating Monticello changes with Jet

We present a qualitative assessment of our approach. As a case study, we considered the integration of the latest changes of the Squeak version of the Monticello core package into the Pharo system. Our case study consists in two parts. In the first part, we asked one of the *integrators* of Pharo to use the JET tools while integrating parts of Monticello. For the second part, we invited a *developer* knowledgeable about the Monticello core package to use JET to *estimate the effort* required for integrating the Squeak branch of Monticello with Pharo. While the former part of the case study provides us some insights into the perceived usefulness of the different features of JET, the latter part aims at assessing the effort and time required to use JET for analyzing the history of Monticello in Squeak.

Note that this case study does not allow us to make any generalizable claims regarding the usefulness of our approach. Given the challenges associated with change integration, a full-fledged validation would require a controlled experiment with *advanced* developers (instead of for example groups of master students). Such an experiment lies outside the scope of this paper and is considered future work.

6.1. Case study description: Monticello versioning system

After forking from Squeak in 2008, Pharo developers modified their own branch of the Monticello core package (267 commits), while the Squeak developers continued the development of the original core package (196 commits). While some of the changes in the Squeak branch were already integrated into Pharo, this process occurred in an entirely ad-hoc manner. The Monticello core package implements the versioning system used by Squeak and Pharo. Table 1 shows the size of Monticello in Squeak and Pharo.

Prior to performing the case study, we loaded the history of the Squeak branch of Monticello (from February 2007 to April 2012) into JET.[9] Table 2 gives an overview of the number of extracted changes and different kinds of deltas characterized by JET. Note that we were not able to load 3 versions of Monticello, as these were missing from the repository.

6.2. Integrator experiences

As mentioned earlier, the first part of our case study consisted in an experienced Pharo integrator using JET while integrating changes from the Squeak branch of Monticello into Pharo. The integrator was also given access to the tools of Monticello (browsing the repository, checking diffs) to confirm the information he obtained from JET.

6.2.1. Protocol

We observed the integrator while using JET during 5 sessions of 30 minutes each. During these sessions we asked him to talk out loud which made it easier to take notes of his actions. After each session, we asked him for some clarifications about certain choices he made while using the tools. On average the integrator analyzed over 12 deltas per session. According to the integrator, his usual rate for such a task is about 5 to 6 deltas over the same period of time. While this is encouraging, we cannot claim that this speed-up was caused by the use of our tools. It could also have been due to other factors such as the complexity of a change.

[9] An image containing JET and the case study can be found at: http://soft.vub.ac.be/~vuquilla/JET-Pharo-1.3-13328-OneClick.

The integrator produced a log for each delta in the list. He wrote a little summary and some notes about the difficulty of integrating each delta and what should be done: if the changes were already integrated, if the changes were applicable to Pharo, if the changes were still valid (not modified later in the list), and so on.

6.2.2. Results

Identifying change authors The integrator was acquainted with the level of expertise of certain committers. Therefore he took more time to analyze the changes made by not so experienced developers. From this perspective, having the name[10] of the committer associated with a number for identifying a delta was considered a useful feature of JET.

Prioritizing deltas The integrator started the analysis of the case study by prioritizing the deltas to be integrated based on their complexity. To this end, he based himself on the colors identifying kinds of deltas. The integrator identified that the *islands* (pink) and *sources* (green) deltas were the most suitable candidates to integrate. As *islands* only contain changes without dependencies, he considered these to be easy to integrate and ignored them at the beginning. When asked for the reason, he explained that his motivation was to spend his efforts on the changes that were more complex and thus more challenging to integrate. He started with investigating the *source* deltas in more detail. In particular, he wanted to identify the different 'chains' of deltas within the stream of changes that might constitute a single feature or fix. As such chains originate from a *source*, the integrator ignored the *intermediate* (orange) and *end* (gray) deltas for now.

Afterwards, the integrator mentioned that the colors of the nodes were useful in providing an initial assessment of the kinds of deltas, and that the consistent use of the color conventions eased usage of the tool. Despite the presence of the JET map, we noticed that the integrator mostly used the dashboard (list view). We hypothesize that this is because the JET map of the case study was rather complex and the layout algorithm did not succeed in providing an intuitive layout.

Using change metrics The integrator frequently used the metrics of the changes in combination with the query browser. For example, when he noticed that for a particular change the altered entity was also changed in later deltas, this often served as a cue to open the query browser and inspect the evolution of the changed entity. We identified three different usages where the presence of change metrics supported the integrator. First, the integrator used the metrics to identify if a particular method should be integrated by checking whether it was called anywhere later on in the stream. Second, he used the information provided to see if methods were still modified later in forthcoming deltas. As a reason, he mentioned that he did not want to integrate changes that would be superseded by other changes. Third, as the JET tools were loaded in the image in which the integration process was happening, he used the metrics to see if a changed method was already in use in the current version of Pharo.

To illustrate this use of JET, we briefly discuss an example. Somewhere in the history of Squeak, a method fasterKeys was introduced in the implementation of the Dictionary class as an optimized version to return the keys in a dictionary. Consequently, within the Squeak branch of Monticello a method named provision was changed to make use of this optimized method. The change metrics for method provision, as shown by our tool, were *Ch. 4 / 2* meaning that this method changed 4 times in total, of which two times in later deltas than the delta in which the use of fasterKeys was introduced. As the method fasterKeys was not present in Pharo, the integrator was wondering whether this method should also be integrated in order to support the changes made to Monticello. By knowing that the provision method was still changed 2 times in later deltas, he was encouraged to first investigate the evolution of the method. As a result, the integrator noticed that the use of fasterKeys was later on reverted, and that the change could safely be ignored.

Comparing with the current version in Pharo The final feature of JET that the integrator used frequently was the *Working copy diff* to assess the difference between a method in the stream and the current version in Pharo. In addition, he compared the latest version of the method in the stream with its previous or future versions within the stream. The idea was to assess (1) if the change was already in Pharo, and (2) if it was worth to look at this particular version of the change.

Ignoring potential delta dependencies Our characterization of delta dependencies makes a distinction between *needed* dependencies and *potential* dependencies to take the uncertainty introduced by *e.g.* polymorphism into account. The integrator was confused by *potential* dependencies and decided to ignore them, due to the fact that this introduced quite a few false positives to be processed. Even though JET provides support for handling and filtering potential dependencies, this is a clear indication that this feature of our approach should be improved.

6.3. Effort estimation by a developer

6.3.1. Protocol

The second part of our case study focuses on providing insights into the time and effort required to use JET to analyze a stream of changes. We asked a *developer* knowledgeable about Monticello to assess the complete sequence of changes from the Squeak branch of Monticello, and for each delta, determine the potential actions to be taken by someone who wants to

[10] In the Squeak community authors are identified by their initials.

Table 3
Delta classification of Monticello.

	# deltas	Total time (seconds)	Δ average time (seconds)
Already integrated	27	1620	60
Ignore	39	2145	55
To integrate	33	4620	140
Unresolved	35	6300	180

integrate that delta into Pharo. More specifically, we asked him to classify the deltas in several categories: *Already integrated* – meaning that the change was already integrated in Pharo; *Ignore* – meaning that it is not relevant or interesting for Pharo; *Unresolved* – meaning that after investigation it is not clear what decision should be taken; and *To integrate* – meaning that the delta is worth integration and that its impacts are understood and appear to be under control.

For each delta that he processed, we measured the amount of time that he took to assess the delta and decide on its categorization. The developer used a dual-screen setup (27 inch main monitor + 13 inch laptop screen): due to the amount of information provided by JET it requires a significant amount of screen-estate; this setup allowed him to separate the JET tools from his code browsing activities. Next to a Pharo image with the JET tools loaded, he had also access to the Squeak system in order to explore the context of the original changes.

6.3.2. Results

During the time slot of 4 hours that he allocated for the case study, the developer was able to analyze 134 of the 193 deltas. The developer processed the deltas in chronological order, hence starting with the oldest version. He was left to perform his tasks without interference of the authors.

Table 3 gives a summary of the delta classification made by the developer, along with the total time necessary for each category of deltas, and the average amount of time per delta. Next to these average times, we would like to report that there were four *unresolved* deltas that took significantly longer to process than the other deltas (approximately 10 minutes each). Examples of these are the deltas *31.ar* in which trait support was introduced in Monticello, and *154.cmm* in which extensive renaming occurred. As these deltas introduced incisive changes to Monticello, it is not surprising that processing them took a relatively long amount of time.

We would like to mention that this case study does not provide any claims regarding the correctness of the classification as produced by the developer, but merely serves as a means to analyze the amount of time needed to understand deltas using JET. For future work, we plan to use the results provided by the developer and integrate them into Pharo as a means to calculate the amount of false positives.

Overall, the developer was able to classify the deltas in a short amount of time. As expected, the cases marked as *Ignore* took little time, as most of the time such cases were features that are either not applicable to Pharo, or that reversed a previous (incorrect) commit. Likewise, cases identified as *Already integrated* were also processed rather quickly. The reason for this, as mentioned by the developer, is that the dashboard includes a view that allows to compare the difference between a change to an entity and the current version of that entity in Pharo. Consequently, after a few glances the developer could identify that the changes were already integrated and no further investigation of the delta was needed. After the case study, this led us to believe that such cases could be identified (semi-)automatically, which we consider as a possible extension.

While the *To Integrate* and *Unresolved* cases took considerably longer to analyze, the amount of time per delta was on average limited to around 3 minutes. We hypothesize that this is caused by the fact that the number of cases for which the developer needed to invest a lot of time was rather limited. First, most of the deltas did not contain a lot or complex changes. Second, when the same entity was modified in multiple deltas, the developer had to investigate only one change and then could use the query browser to study the evolution of the entity, resulting in that he had to spend less time on subsequent changes to the same entity. Typical examples of this case are API changes, or reverting to prior changes. Third, since deltas tend to be related, the developer could spend a considerable amount of time understanding particular deltas; subsequent deltas that were related to this delta were then processed much quicker.

After the case study, the developer also made a number of observations regarding his process. First, he remarked that the size of the delta is not correlated with the complexity of the analysis required to take a decision. For example, changes to a single polymorphic method could be harder to assess – due to their impact on the system – then a large set of simple changes. Second, the developer remarked that solely analyzing the dependencies of a change did not suffice in order to classify a delta. As one example, he listed the case in which the order of calls in a method were changed. While such a change does not have an impact on the dependencies of the change, it can have a drastic impact on the behavior of the system. In such cases, the developer appreciated the presence of the query browser of JET that allowed him to explore the evolution of the method within the stream. To address this problem our future work is to add an impact model to present the impact taking into account the semantics of the language used.

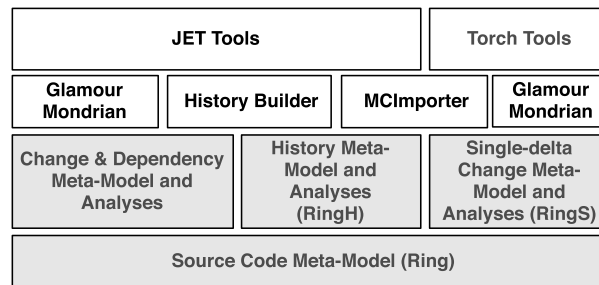

Fig. 20. Elements of JET implementation.

7. Implementation

The approach described has been fully implemented in Pharo. The general architecture follows the process depicted by Fig. 2. Fig. 20 presents the JET elements. JET uses Glamour – a tool-builder library – and Mondrian – a scripting graphical library [3]. In addition JET takes advantage of the Ring meta-model, a infrastructure consisting of three packages (33 classes). The Ring history model is composed of 29 classes, and two extra packages (31 classes) implement the history extraction and Monticello import. Finally JET in itself is composed of 2 packages and 17 classes.

Discussion The implementation of JET was driven by several points: the independence to the code history importer and the source code meta-model.

1. *Code history importer.* The input for the code history can be replaced. Currently the importer takes Smalltalk code versioned with Monticello but nothing in the approach is limited to this. It is possible to build an importer for another versioning system such as SVN or Git. We used Monticello because we had case studies and integrators for projects managed with it.
2. *Metamodel.* The source code meta-model is close to the language independent FAMIX meta-model [8]. The part related to Smalltalk is minimal: we model explicitly pool dictionaries (kind of constants groups), traits (groups of methods) and class variables (static fields). The dependency analyses do not rely particularly on such aspects and are generic because they are based on the relationships we mentioned early (accesses, references, inheritance and invocations). For example, making JET analyzing Java requires to take into account method modifiers when computing method invocations, attributes modifiers when computing field accesses, package structure when computing class references. Such analysis is usually done by a Java fact extractor. The JET dependency analysis is based on dependencies computed on top of the relationships extracted from the source code and from that perspective changes for porting JET to other languages should be limited.

8. Discussion

The outcome of our case study encourages us to believe that JET improves the understanding of changes within their context. Nevertheless, we plan to address a number of JET's limitations in future work:

Other languages While JET has been developed to support Smalltalk code, the underlying analysis is language-independent as is the code model [23]. Therefore applying the same approach to other languages such as Java or C# should not be a problem. In fact, we believe that being able to rely on a static type system can really improve the analysis especially for reducing the set of potential dependencies. But such aspect is not related to JET in particular but to the precision of code fact extractors which produce fuzzier information with dynamically-typed languages than strongly-typed ones.

Impact of changes While our approach provides integrators with more information regarding changes, and the three-way merge algorithm prunes unnecessary changes, we do not provide guarantees that the code will execute when integrated. In fact, semantic merging is still an open challenge. This point is even more challenging for dynamically typed languages where static analyses are limited and the code model is less precise. Still tools should be able to show the potential impact that a change may have on the current system. In future work, we will investigate the use of program slicing techniques on both source code and changes to provide a fine-grained impact analysis. We would like also to understand if it is possible to compare the impact of the change in the source vs. the impact on the target. In particular managing the amount of information and how to present it to the end-user in an adequate form is also a challenge.

Cross-branch integration Even though JET supports the analysis of streams of changes, it currently does not provide a full-fledged solution for assessing the impact of a stream of changes on a target system and for migrating changes from one branch to another. For example, in our validation the integrator performed a dependency analysis of the changes made to

the Monticello system in the Squeak environment without taking into account the evolution of Monticello in Pharo. The integrator only looked at the current version of Monticello in Pharo without considering some other versions in its history. We plan to extend JET such that the history of multiple systems can be taken into account.

9. Related work

To the best of our knowledge, no related tools nor approaches that aim at supporting understanding commits with the goal of merging such commits across branches exist. However, there are a number of related approaches focusing on representing, replaying, characterizing, analyzing and understanding changes.

Modeling change A vast body of work exists on meta-models that provide common representations leveraged by various software engineering tools. Next to meta-models, such as FAMIX [8], that focus on the representation of a single version, there are approaches for modeling changes and multiple versions of a system. A good overview of this research can be found in the book chapter by D'Ambros et al. [6]. HISMO [12] is an extension to the FAMIX meta-model that allows for the representation of multiple versions of a system. For each version in the history, a complete model of that version – along with information that relates source code entities over various versions – is stored. Although HISMO models can easily be imported from a Monticello repository, we did not reuse this model as it does not provide a fine-grained representation of changes.

SpyWare [21] records all changes that are made to a system using the integrated development environment (IDE). Internally, SpyWare provides a fine-grained model where each individual change to the system is stored. CheOPS [9] offers a similar meta-model for representing and storing changes. Another similar approach for the reification of changes is the one taken by Syde [13], a tool that logs the changes made by several developers in parallel. While these approaches do provide a fine-grained, first-class representation of changes, these changes do not allow these changes to be constructed from the history of an existing system.

Fine-grained patching Semantic patches [18] offer a declarative domain-specific language (SmPL) for expressing collateral evolutions. The idea behind SmPL is that, rather than creating a patch that is only applicable to a single source code file, a developer can describe a generic patch as a transformation of the source code that can be applied to multiple source code files. As an extension of this work spdiff is presented [1]: a tool that, given a set of standard patches, automatically generates a semantic patch.

While semantic patches can be used to generalize a set of changes made in one branch and apply these changes to another branch, they do not fully tackle the issues addressed in this paper. In particular, semantic patches do not aid in solving the problem that a set of changes might depend on previous changes that were performed in the same branch, and that also need to be migrated in order to obtain a functioning system. As our approach aids integrators in understanding a set of changes and its dependencies, JET is largely complementary with semantic patches and can potentially aid integrators in defining and managing such semantic patches.

Collard et al. [5] present an approach for easing the integration of large changes by factoring single commits into a series of smaller changes based on syntactic criteria. Based on a XML representation of a *diff* of a system, a developer can partition this *diff* into a number of smaller sets of changes by querying the XML representation. The idea is that these factored changes can then be integrated individually. First, this process of factoring a commit is done manually and might benefit from the information provided by our tool. Second, similar to semantic patches, the factored commit does not take into account previous commits and therefore does not address the problem of dependencies between changes.

Change characterization Darcs (http://darcs.net) is a distributed change-based source-code management system based on an algebra of patches, named *the theory of patches*, for manipulating changes. This theory is about the commutation, or reordering, of changes in such a way that their meaning does not change. The Darcs merge operation is based on the patch commutation algorithm. Darcs supports *cherry picking*, as also found in Git, allowing users to choose the patches that they want to check in or check out. However, semi-automatic handling of conflicts and merging of features are not well supported.

Dragan et al. [7] propose a technique to characterize a commit based on the methods that were added or removed in that commit. In previous work, they have presented a categorization of methods (stereotypes) that take various properties of the method (accessing data, changing state, interaction with other objects, and so on) into account. Their technique leverages these method stereotypes and, by studying the distribution of the various kinds of method stereotypes within a commit, proposes a number of categories of different kinds of commits. This technique is related to our work in the sense that the identified commit types can provide an integrator with valuable information regarding the size and scope of a commit. However, this technique does not provide any information regarding the dependencies between commits and the ease with which a commit can be integrated across branches.

Change impact analysis/change dependencies Dependencies between changes have been used in the context of change impact analysis. Chianti [19] decomposes the difference between two versions of a Java system into a set of atomic changes. Change impact is then reported in terms of affected (regression or unit) tests whose behavior may have been modified

by the applied changes. Chianti relies on syntactic dependencies between atomic changes for the change impact analysis. Other approaches extend Chianti and use dependencies for similar change impact analyses. Ren et al. [20] extended the syntactic dependencies to three kinds of dependencies between atomic changes that capture syntactic and partially semantic dependencies to detect failure-inducing changes between two versions. While the dependencies provided by Chianti and its derived approaches overlap with our change dependencies, they only apply to a single delta. These approaches do not offer characterization of deltas based on change and delta dependencies within a stream of changes.

CGIs [11] determines the impact of historical code changes on a particular code segment by means of dependence graphs. This approach guides developers to investigate failures in unchanged functions that are affected by bugs introduced in prior code changes. Structural dependencies between C functions are used to build the dependence graphs. These dependencies correspond to a subset of our change dependencies. GENEVA [14] uses dependencies to perform change impact analysis for providing recommendations to developers (*e.g.*, predicting long-term change coupling). This approach builds change dependency graphs (known as change genealogies [4]) by ordering changes based on dependencies, and later applies model checking to the change genealogy. The dependencies are determined across transactions in version archives. While these dependencies are very similar to our change dependencies, GENEVA's change genealogy and CGIs do not offer the notion of deltas and dependencies between deltas that can be used to characterize sets of changes within a stream. Moreover, both determine dependencies that are not relevant in the context of integration.

Understanding change Fritz and Murphy [10] present a study in which they interviewed developers regarding the different kinds of questions they need answered during development. Alongside this study, they introduce the information fragment model and associated prototype tool for answering the identified questions. This model provides a representation that correlates various software artifacts (source code, work items, teams, comments, and so on). By browsing the model, developers can find answers to particular development questions.

While a number of the questions that developers need answered during development aligns with those they need answered during integration of changes, the information fragment model does not provide functionality to calculate dependencies between changes, which is necessary for integrating changes across branches.

The approaches performing change impact analysis presented beforehand provide a means to better understand changes. However, some of them are limited to a single delta, and none of them support understanding streams of changes in the context of integration. JET could be complemented with a change impact analysis similar to the one provided by Chianti [19].

In previous work we built the Torch tool [23]. Torch allows integrators to understand the changes within a single delta. It visualizes how a delta is related to the structure of the system. JET generalizes and augments Torch's philosophy: (1) a stream of changes is characterized, (2) a stream can be queried and navigated, (3) dependencies between the changes are computed and help driving change analyses. Changes are not treated in isolation but within a stream of changes.

10. Summary

This paper presented JET, an approach for characterizing deltas and dependencies within the context of a stream of changes. Next to introducing our model for changes and dependencies, and discussing the algorithm underlying our approach, we introduced a set of tools (*i.e.*, the JET dashboard, the JET map and the JET change query browser) that complement the approach. These tools allow an integrator to visualize and analyze dependencies between deltas. Finally, we performed a qualitative assessment of the capabilities of JET by performing an exploratory case study on a considerable stream of changes in the context of a non-trivial open-source system in operational use.

References

[1] J. Andersen, J. Lawall, Generic patch inference, Autom. Softw. Eng. 17 (2010) 119–148.
[2] S. Apel, J. Liebig, B. Brandl, C. Lengauer, C. Kästner, Semistructured merge: rethinking merge in revision control systems, in: ESEC/FSE, ACM, ISBN 978-1-4503-0443-6, 2011, pp. 190–200.
[3] A. Bergel, D. Cassou, S. Ducasse, J. Laval, Deep Into Pharo, Square Bracket Associates, ISBN 978-3-9523341-6-4, 2013.
[4] I.I. Brudaru, A. Zeller, What is the long-term impact of changes?, in: Int. Work. on Recommendation Systems for Soft. Engineering, ACM, ISBN 978-1-60558-228-3, 2008, pp. 30–32.
[5] M. Collard, H. Kagdi, J. Maletic, Factoring differences for iterative change management, in: Int. Work. on Source Code Analysis and Manipulation, IEEE, 2006, pp. 217–226.
[6] M. D'Ambros, H. Gall, M. Lanza, M. Pinzger, Analysing software repositories to understand software evolution, in: Software Evolution, Springer, 2008, pp. 37–67.
[7] N. Dragan, M. Collard, M. Hammad, J. Maletic, Categorizing commits based on method stereotypes, in: ICSM, 2011, pp. 520–523.
[8] S. Ducasse, N. Anquetil, U. Bhatti, A. Cavalcante Hora, J. Laval, T. Girba, MSE and FAMIX 3.0: an interexchange format and source code model family, Technical report, INRIA, 2011.
[9] P. Ebraert, First-class change objects for feature-oriented programming, in: WCRE, IEEE CS, 2008, pp. 319–322.
[10] T. Fritz, G.C. Murphy, Using information fragments to answer the questions developers ask, in: ICSE, ACM, 2010, pp. 175–184.
[11] D.M. German, A.E. Hassan, G. Robles, Change impact graphs: determining the impact of prior code changes, Inf. Softw. Technol. (ISSN 0950-5849) 51 (10) (Oct. 2009) 1394–1408.
[12] T. Gîrba, S. Ducasse, Modeling history to analyze software evolution, J. Softw. Maint. Res. Practice 18 (2006) 207–236.
[13] L. Hattori, M. Lanza, Mining the history of synchronous changes to refine code ownership, in: Int. Work. on Mining Soft. Repositories, IEEE, 2009.
[14] K. Herzig, A. Zeller, Mining cause-effect-chains from version histories, in: 22nd Int. Symp. on Soft. Reliability Engineering, IEEE, 2011, pp. 60–69.

[15] J. Laval, S. Denier, S. Ducasse, J.-R. Falleri, Supporting simultaneous versions for software evolution assessment, Sci. Comput. Program. 76 (12) (2011) 1177–1193.
[16] T. Lindhom, A 3-way merging algorithm for synchronizing ordered trees – the 3DM merging and differencing tool for XML, Master's thesis, Helsinki Univ. of Technology, 2001.
[17] T. Mens, A state-of-the-art survey on software merging, IEEE Trans. Softw. Eng. (ISSN 0098-5589) 28 (5) (2002) 449–462.
[18] Y. Padioleau, J. Lawall, G. Muller, Documenting and automating collateral evolutions in Linux device drivers, in: EuroSys, 2008, pp. 247–260.
[19] X. Ren, F. Shah, F. Tip, B. Ryder, O. Chesley, Chianti: a tool for change impact analysis of Java programs, in: OOPSLA, ACM, 2004, pp. 432–448.
[20] X. Ren, O.C. Chesley, B.G. Ryder, Identifying failure causes in Java programs: an application of change impact analysis, IEEE Trans. Softw. Eng. 32 (9) (Sept. 2006) 718–732.
[21] R. Robbes, M. Lanza, SpyWare: a change-aware development toolset, in: ICSE, ACM, 2008, pp. 847–850.
[22] V. Uquillas Gómez, Supporting integration activities in object-oriented applications, Ph.D. thesis, Vrije Universiteit Brussel, Belgium & Université Lille 1, France, Oct. 2012.
[23] V. Uquillas Gómez, S. Ducasse, T. D'Hondt, Visually supporting source code changes integration: the torch dashboard, in: WCRE, 2010, pp. 55–64.
[24] V. Uquillas Gómez, S. Ducasse, T. D'Hondt, Ring: a unifying meta-model and infrastructure for Smalltalk source code analysis tools, Comput. Lang. Syst. Struct. 38 (2012) 44–60.
[25] T. Zimmermann, P. Weißgerber, Preprocessing CVS data for fine-grained analysis, in: Int. Work. on Mining Soft. Repositories, IEEE, 2004, pp. 2–6.