Tracking dependencies between code changes: An incremental approach
Lucas Godoy, Damien Cassou, Stéphane Ducasse

To cite this version:

Lucas Godoy, Damien Cassou, Stéphane Ducasse. Tracking dependencies between code changes: An incremental approach. IWST 2014, Aug 2014, Cambridge, United Kingdom. hal-01076238

HAL Id: hal-01076238
https://hal.science/hal-01076238

Submitted on 21 Oct 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Tracking dependencies between code changes: An incremental approach

Lucas A. Godoy
INRIA Lille-Nord Europe UBA
lucas.ariel.godoy@inria.fr

Damien Cassou Stephane Ducasse
INRIA Lille-Nord Europe
damien.cassou@inria.fr stephane.ducasse@inria.fr

Abstract

Merging a change often leads to the question of knowing what are the dependencies to other changes that should be merged too to obtain a working system. This question also arises with code history trackers – Code history trackers are tools that react to what the developer do by creating first-class objects that represent the change made to the system. In this paper, we evaluate the capacity of different code history trackers to represent, also as first-class objects, the dependencies between those changes. We also present a representation for dependencies that works with the event model of *Epicea*, a fine-grained and incremental code history tracker.

Keywords change propagation, IDE, history, dependency analysis, software evolution

1. Introduction

Software systems evolve in response to change in their functional requirements. These changes made through time to the source code of software systems is what we call their code history. We can keep track of this evolution process through the usage of Version Control Systems (VCSs) such as Git[1].

Since software engineering is part of software evolution [RL07], a development environment that represents changes as first-class entities that can be referenced, queried and passed along in a program [EVC$^+$07] is fundamental for a *change-oriented* engineering approach. This cannot be accomplished using the mainstream VCSs in use today for the following reasons:

- The semantic information of the changes made to the system is scattered in a large amount of text, so tracking entities involves parsing several versions of the entire system.
- Several independent fixes and features can be introduced in one single commit, making it hard to differentiate them.
- The time information of each change is reduced to the time when each commit is performed, so all information about the exact sequence of changes which led to these differences is lost.

To minimize the effort for sharing and merging code through a VCS, some best practices have been established:

- Commit small, related, self-contained change sets. This is what is usually known as an *atomic commit*[2].
- Usage of a descriptive commit message.
- Commit regularly.

Following these best practices requires a lot of discipline. As a result, committing unrelated changes happens regularly in software development. This means that either the tools are used do not allow to follow the best practices or the effort to follow the aforementioned best practices is too high for developers.

To reduce this effort, a new generation of tools (that we call *code history trackers*) was born. These tools are conceptually *event-based*: they react to what the developers do by creating first-class objects that represent the changes made to the system. Remarkably Smalltalk change tracking systems (ChangeSorter) is one the elder code history trackers and it predates mainstream versioning systems.

However, we consider that none of the current code history trackers has a minimal set of desired features to reduce the effort required to do an atomic commit. The most important of these features is the ability to detect dependencies between changes made to the system, or what we call *dependency tracking*.

In general, the *re-assembly of changes* has been historically supported through a feature called *cherry-picking*. The support for cherry picking enables programmers to extract incremental improvements that are spread over a set of many changes. Consider for example that a task has involved a refactoring that the programmer must manage and share as a separate improvement. Programmers can first have a look at the list of all versions to identify both the individual changes that constitute the refactoring and the version from which the main task started.

Over time, it becomes increasingly difficult and tedious for a developer to determine whether a change from another branch or fork can benefit the system, which makes it difficult and time consuming. This difficulty is emphasized by the lack of support for the analysis of dependency between changes. Indeed it is rare that a change happens in isolation.

There is a need for tools that can detect dependencies automatically, so the programmer doesn't need to remember these dependencies or to identify and select them manually.

The contributions of this paper are:

- An evaluation of current history tracking tools for Smalltalk and how they facilitate the dependency tracking to reduce the human effort needed to follow the described best practices.
- The definition of one model and the building of a dependency tracking mechanism on top of it, focusing on simplicity, to assist the programmer in the process of re-assembling changes. Given the dynamic nature of Smalltalk, the approach is not completely accurate for message sends. And it is not fully automatic, since the developer has always the chance to edit the suggestions of the tool.

[1] `http://git-scm.com/`

[2] `http://en.wikipedia.org/wiki/Atomic_commit`

Table 1. Evaluation summary

System	First-class objects	Incremental	Dependencies	Refactors	Exploration
ChangeSet	Partial	✓	✗	✗	ChangeSorter
Ring	✓	✗	✓	✗	Jet
Epicea	✓	✓	✗	✓	Log Browser
CoExist	✓	✓	✗	✗	Version Bar

2. Related work

In this section we describe three existing tools for code history tracking and we evaluate how they assist the developer to facilitate the best practices listed earlier.

2.1 Evaluation criteria

To evaluate the existing tools we consider the following questions:

- Are changes modelled as first-class objects?
- *Is it incremental?* Is it possible to analyze a single change or does it need to create a full history log? Incrementality makes the semantic representation of the model easier to mantain.
- Are dependencies between changes modelled?
- Are high-level refactorings modelled?
- Does the solution provides a flexible way to explore the list of changes?

2.2 The Smalltalk ChangeSorter/ChangeSet

The traditional Smalltalk ChangeSet log is a reliable mechanism to log the source modifications immediately after any editing operation happens on an image [Gol84]. It may be used as a recovery tool by backtracing to the most recent non/erroneous state of the image and reapplying changes listed by the ChangeList tool.

However, the changes are written to a log file as executable statements and only classes and methods are modelled as first-class objects. Additions and removals of attributes can only be detected by comparing different versions of the program. These records have no information about high-level changes as refactorings and mix source management with the events that make the system evolve from one state to another. As a result, not all events can be recorded, the granularity of the events is too coarse and the exploration of the change list made with the ChangeSorter is cumbersome and error-prone.

Considering these limited the representation of changes, is no surprise that the model does not include a representation for dependencies between changes.

2.3 Ring

Ring [UGDD12] is a unified source code meta-model that:

- Has a common API with the Runtime and Structural Smalltalk model.
- Represents every program entity as a first-class object. Unlike the standard Smalltalk model, it can represent variables as objects instead of strings.
- Serves as the underlying meta-model for the history and change meta-models.

The history meta-model, called *RingH*, models source code entities such as packages, classes, methods and atributes as well as the relationships between such entities such as class inheritance, method call, class reference and attribute access. The history models are extracted from the source code history contained within versioning repositories.

RingC is the change and dependency meta-model, which uses the information contained within the RingH model and creates sets of changes (instances of class `RGChange`) for each snapshot retrieved from the repository. These sets are called *deltas*. When there is a dependency between two RGChange objects, the RingC model creates a `RGChangeDependency` that represents it. This dependency can happen within changes in the same delta or between changes belonging to different deltas. If a reference to a non-existing object is introduced in a change, that change has an external dependency that is modelled with a stub class.

Even when the object model improves the granularity of the events recorded, the Ring approach is still unable to detect the most high-level events (i.e: refactorings).

Jet [UG12] is a semi-automated tool built on top of RingC that offers a characterization of the changes and dependencies within a stream of changes. It is not incremental, since it creates a full history log by extracting information from repositories instead of reacting in real-time to the changes made by the developer. Because of this, the process of importing the repository data to generate the history and then extract the changes and their dependencies can be time consuming for big projects.

2.4 CoExist

CoExist [STCH12] is a tool for Squeak/Smalltalk that relies on the idea of continuous versioning: any change made to the system triggers the creation of a new version storing the change as well as a complete snapshot of the current system. Unlike a traditional VCS that store the source code changes in separate files, these snapshots are internal data structures that store the state of the system in a particular point in time.

The user can go back and forth between versions using the *Version Bar* and create additional working environments to inspect, modify and debug versions of interest. The system also allows to run tests continuously, to collect results for every individual version and to run a potentially new test on previously created program versions.

Despite these features not found in the other tools, CoExist is not free of limitations. Some classes cannot be versioned and switching between versions requires a restart of the application under development. It also lacks support for direct references to class objects and its model does not include dependencies between changes.

2.5 Epicea

Epicea [DCD13][3] is a code history tracker built on top of the Ring core that represents the changes in entities with events. Each one of these events contains two snapshots representing how the entity was before and after the change. This capacity of reacting to the events as they happen provides the exact sequence of changes that led to the differences between each pair of snapshots. It is also easier to maintain a semantic representation of the model, requiring code parsing only at the method level.

The event model has some trade-offs between accuracy and simplicity. For example every time Epicea detects a change in a

[3] `http://smalltalkhub.com/#!/~MartinDias/Epicea`

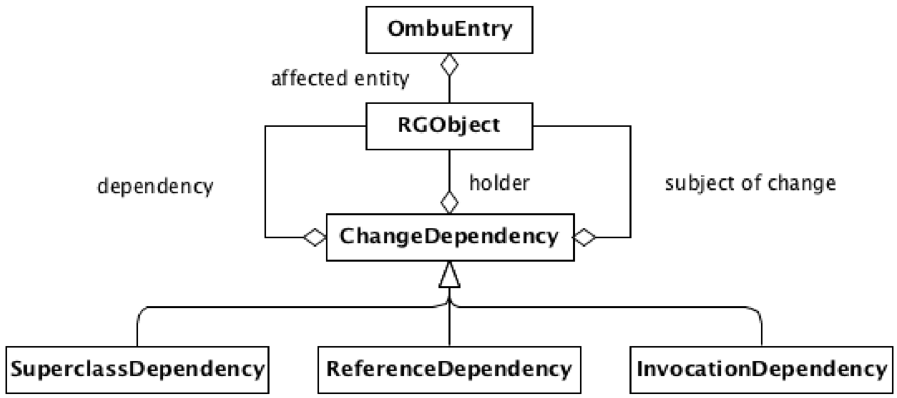

Figure 1. Object model

class, it is unable to distinguish between an addition or removal of an instance variable and the addition or removal of a class variable. This is not a major drawback for its current features, but it is something that will have to be considered if we want to add dependency tracking to its feature set.

Epicea writes each event immediately to disk using one Ombu file per session instead of a single ChangeSet file, making easier the recovery of the exact sequence of changes that originated the differences between the snapshots of the affected entity. It also can export the log entries to a ChangeSet file (only for events supported by the ChangeSet format).

Unlike the standard ChangeSet model and the complete Ring model, Epicea events can represent high-level refactorings. This simple event-based model replaces the RingH layer of the Ring ecosystem but there is no model to represent dependencies between the changes triggered by those events.

Also the Log Browser makes easy to go back and forward in time using the events logged, leading to an easier exploratory development.

3. Additions to the Epicea object model

Our objective is to create an object model to represent dependencies between the existing change model of Epicea. In this section we define what a dependency is, how to represent a dependency with an object and how to extract the dependencies from each changed entity in the system.

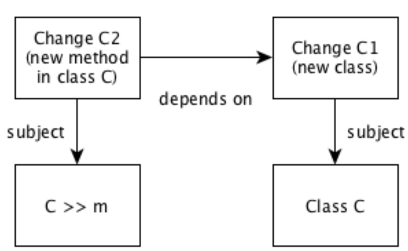

Figure 2. Dependency in a method creation

A change is always applied to a *subject*. *Creational changes* are changes which have as subject a new entity that they produce. In this case, a change `c1` is said to depend on a change `c2` if that is the creational change of the subject of `c1` [Figure 2]. For example, methods can only be added to existing classes.

Also, the source code of `m` can contain references to other entities and messages sent [Figure 3]. The entities referenced in the source code of `m` must exist to ensure its compilation and proper execution. Because of this, we need to parse the source code associated to every change.

Therefore, our dependency object [Figure 1] will be composed of three references to entities:

1. The subject of change or entity to be modified.
2. Optionally, a class holder. This is the class that holds the subject of change. It will be nil for classes, since they don't need a holder.
3. Optionally, a set of dependencies extracted from the source code. If the event is a class addition or any entity removal, it will be empty.

This set of dependencies is generated from the source code of methods. In the next subsections we explore the different dependencies that we can find.

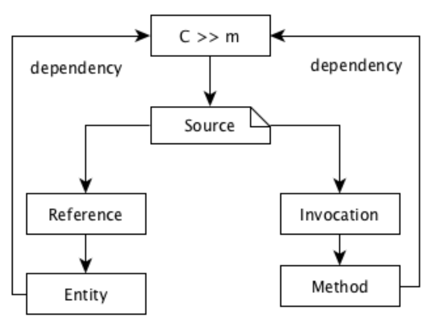

Figure 3. Dependencies extracted from a method

3.1 Types of dependencies

We have three types of dependencies between entity changes. In all cases, parsing the code associated to the entity is needed.

- *Class hierarchy dependencies*: for each change in a class, which can be a change in a method or in the class definition, the superclass must exist. The same happens when self is called from the code of the method that is the subject of change.

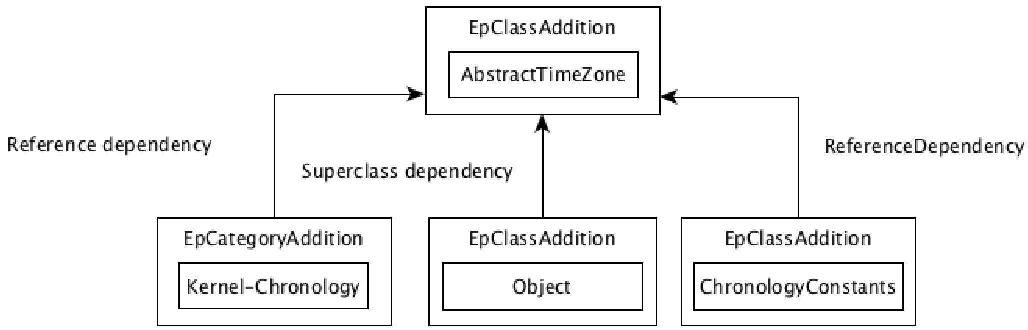

Figure 4. Dependencies of a class addition

- *Reference dependencies*: they are references to temporary, instance and class variables in the source code of any method. Also references to classes.
- *Message sends*: messages sent in the source code of any method or expression evaluation. Since Smalltalk is dynamically-typed, in absence of type information and presence of polymorphism, there is a need to provide very fine-grained information about messages sent to find dependencies in an accurate way.

It may happen that a dependency for a change is located in a different package. We call this an external dependency. And if the dependency doesn't exist in the system, we call it a missing dependency. Both cases will have their first-class object in our model.

```
Object subclass: #AbstractTimeZone
       instanceVariableNames: ''
       classVariableNames: ''
       poolDictionaries: 'ChronologyConstants'
       category: 'Kernel-Chronology'
```

Listing 1. Class definition example

Listing 1 shows the code of the class AbstractTimeZone. This class inherits from Object, uses the pool dictionary ChronologyConstants and is located in the category Kernel-Chronology. So we can extract 3 dependencies from this definition [Figure 4]:

1. The class Object must be defined.
2. The shared pool ChronologyConstants must be defined. This also means that the class SharedPool must be defined.
3. The package Kernel-Chronology must be defined.

```
Trait named: #TClass
       uses: TBehaviorCategorization
       category: 'Traits-Kernel-Traits'
```

Listing 2. Trait definition example

Trait definitions are similar. The dependencies in Listing 2 are the Trait class, the TBehaviorCategorization trait and the category.

3.1.1 Message sends

If a message is sent inside the code of a method, we can look for the methods that potentially will receive the call (i.e., dynamic dispatch). This is what we know as a *candidate set* [DAB+11]. Since candidate sets can contain false positives, we categorize message sends as follows:

- Messages sent to self: all candidates for the call need to be in the hierarchy tree of the class in which the method is defined. This case can lead to false positives when the method is declared in many classes that belong to same hierarchy.
- Messages sent to super: this corresponds to the super calls within a method, which is bound statically. So it must be defined in a direct or indirect superclass in which the method is defined.
- Messages sent to classes: The receiver of this message is a class reference.
- Unknown sends: the call of the receiver is unknown, so the candidate set consists of all methods with the given selector. This case can lead to false positives.

```
1 AbstractTimeZone >> printOn: aStream
2        super printOn: aStream.
3        aStream
4                nextPut: $(;
5                nextPutAll: self abbreviation;
6                nextPut: $).
```

Listing 3. Method definition example

Listing 3 shows the code of method printOn: from the class AbstractTimeZone. We can extract many dependencies from this change [Figure 5]:

1. First of all, we need the class AbstractTimeZone to add the method.
2. At line 2, we have the message printOn: sent to super, so this depends on Object ≫ printOn: or ProtoObject ≫ printOn: according to the class hierarchy of AbstractTimeZone.
3. At line 5, we have a message send to self. So this depends on a method called abbreviation that can be on any member of the current class hierarchy.
4. Starting at line 3, we have many messages sents to a parameter. Since we don't know to which class the parameter belongs, this is an unknown invocation. It's candidate set are all the methods called nextPut: and nextPutAll:.

Listing 4 shows the code of AnnouncementSpy ≫ buildList. Since we are sending the message new to the class PluggableListMorph, we are sure that this class and its method must be defined. We could assume the result of this message is an instance of PluggableListMorph, but since we cannot be sure that new returns an

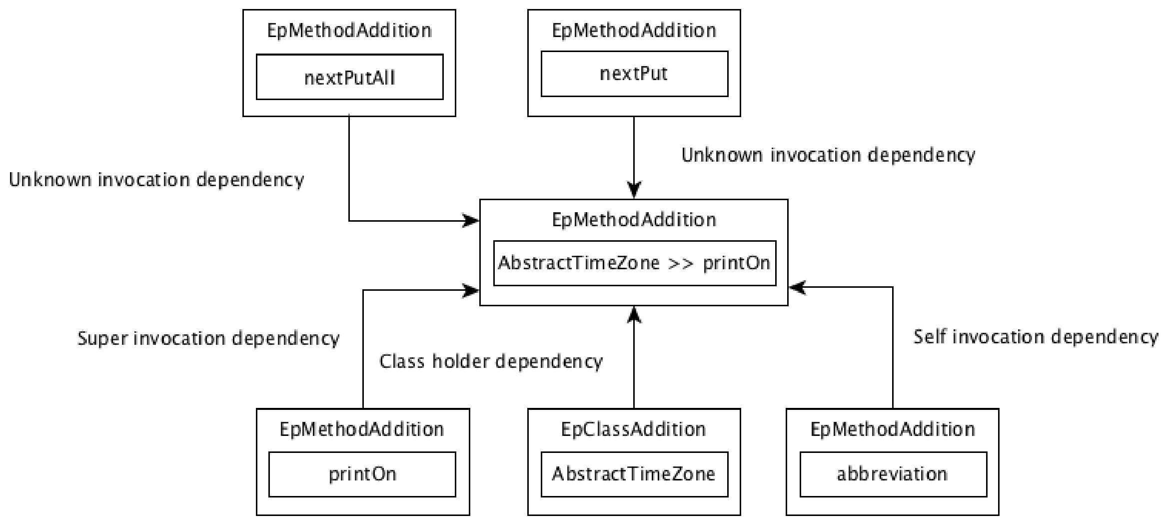

Figure 5. Dependencies of a method addition

instance of the class, we're forced to look for all the implementors of the message.

```
buildList
	^ (PluggableListMorph new)
		on: self
		list: #announcements
		selected: #index
		changeSelected: #index:
		menu: #buildMenu:
		keystroke: nil.
```

Listing 4. Example of a message sent to a class

3.2 Unknown message sends with self

Let's suppose we added the method addAll: to the class Collection [Listing 5]:

```
addAll: aCollection
	aCollection do: [:each | self add: each].
	^ aCollection
```

Listing 5. Unknown message sends with self

Among others, we have a dependency with add:. It's code is shown on Listing 6.

```
add: newObject
	self subclassResponsibility
```

Listing 6. Role of subclassResponsibility

This one has a dependency with subclassResponsibility. But this is not enough to make addAll: work. We should include all the add: messages in the Collection hierarchy. This is a case where a dependency found in a message send to self can have false positives.

3.2.1 Reference dependencies

Let's illustrate how to handle variable references by looking at the code of this method in the class OrderedIdentityDictionary [Listing 7].

We have messages sent to self and super, that we already covered. We have an unknown send for the message key, that has more 13 implementors in a standard Pharo 3.0 [4] image. We also have an unknown sned for ifFlase:, but in this case there are only 3 implementors restricted in the Boolean class hierarchy.

```
add: anAssociation
	(self includesKey: anAssociation key)
		ifFalse: [ keys add: anAssociation key ].
	^ super add: anAssociation
```

Listing 7. Reference dependency example

We also had a reference to a variable called keys. There is no temporary variable declared in the source code of the method, so it must be an instance or class variable. We said in Section 2.3 that Epicea cannot distinguish between diferent kinds of variable changes in a class definition. This means that we'll have a RGClass-Definition entity that contains a class or instance variable as a dependency. Now a question is raised: which class definition is the one that contains this variable?

```
atRandom: aGenerator
	| rand index |

	self emptyCheck.
	rand := aGenerator nextInt: self size.
	index := 1.
	self do: [:each |
		index = rand ifTrue: [^each].
		index := index + 1].
	^ self errorEmptyCollection
```

Listing 8. Local variable reference

The answer is that this variable should be defined in the last event containing a class definition for A, otherwise the code would not compile (unless the compiler decides to skip compiling for some reason). The worst case would be when the class was created before the installation of Epicea. if this happens, it will scan the full log only to find that the dependency is missing.

[4] http://pharo.org/

In listing 8 we have an example of temporary references, extracted from the method atRandom: of the class Collection.

Since rand and index are defined in the same method, there is no dependency. We could think that there is a dependency with the method itself, but compilation is not possible without the declaration of these two variables.

3.3 Modification and removal of entities

Modification of entities work in a similar way to what we already explained. The only difference is that modification events have two Ring entities (the subject of change and the result of the change) instead of only the subject. The process of dependency extraction is the same, but in this case is the code of the result of the change that will be parsed.

For deletions, the only events we consider as dependencies are deletions of entities held by a deleted holder. For example, if a method m from class C was deleted and then class C was also deleted. The deletion of m will be added to the candidate set of the deletion of C.

4. Implementation details

4.1 Anatomy of an Ombu entry

An entry in an Ombu file has a content, which can be any object, and a dictionary of tags. In the specific case of Epicea, the content is a change event [Figure 6]. The tag dictionary is used to store metadata like the author and the time of the change.

Another thing that is stored in the tag dictionary is the prior reference. Each entry has a reference to the prior change and this is the way the changes are linked as a list. We can use this mechanism to persist the dependency information for each entry.

As a second step, it is also desirable to be able to get the entries that contain the subject of change for each one of the dependencies.

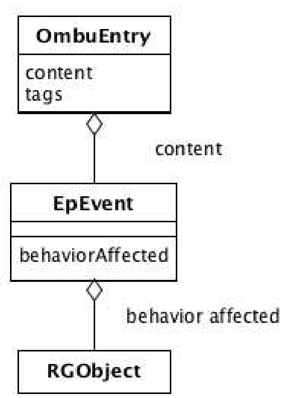

Figure 6. Anatomy of an Ombu entry

4.2 Retrieval of class holders

Epicea events can contain one or two RGObjects. Addition and removal of entries contain the new entity, while modifications contain the subject and the result. We defined the class holder in our model but it doesn't exist as a first-class object in the event. In these cases, the event only knows the name of the holding class. Therefore, we have to look in the log for the event of the creation of the holding class.

4.3 Retrieval of entries containing subjects of change

Dependencies are defined between affected entities: classes, methods and so on. Once the dependencies for an affected entity have been established, we have to find the events that affected those entities.

For example, let's suppose that we modified the definition of a class A. A new EpClassModification event object will be created and it will contain two instances of RGClassDefinition: one that represents the old class definition and another to represent the new one [Figure 1]. We have to find an entry that contains the event with the RGClassDefinition that represents the creation of the class A.

Almost all Epicea events are created with an RGObject as an internal collaborator [5]. Therefore, to have access to those entities, we can keep them in a multimap indexed by selector. The maintenance of the multimap (addition, changing and removal of entries) will be made at the moment of the event creation. And it won't be necessary to scan the complete log to find the related entries.

Since the events and entities are already present in the current Epicea implementation, the only additional objects that will be added are the dependencies.

5. Future work

In this section we describe some improvements to our initial solution.

5.1 Events vs. entries

In the current Epicea implementation, the Ombu entries are the ones that are linked through the prior reference. One of the limitations of this approach is that overlapping entries repeat the code through the related entries. For example, if we have an entry A with a class definition and an entry B with a modification to that class definition, B will contain all the code defined in A instead of having just a reference to it.

Another option would be to move the references to the event level. The model would be more sound from a semantic point of view and we can replace the duplicated code for a reference to the underlying event.

5.2 Visualization of dependencies

Once the extensions for the Epicea model are in place, we can modify the Log Browser to display the relationship between the entries in a graphical way.

One option is to draw lines between the entries in the Log Browser, as the GitK tool does. Another one, possibly more complex, is to add a tab in the lower panel that shows a dependency tree. This approach can be found in m2eclipse [6].

5.3 Performance test and optimization

It is desirable to test the performance in terms of execution time and memory consumption of this new features when they are used with a Log that contains several entries.

One alternative to reduce the memory footprint is to implement the multimap using a *Trie*. The keys will be the entity names, but all entities with a common prefix in their selectors will share that part of the key. The worst-case access time for a given selector would be the length of the longest selector defined in the system.

6. Conclusion

Detection of dependencies between changes modelled as first-class objects are a very important feature of code-history trackers, since it reduces the effort of the developer to perform tasks like atomic commits.

[5] The exception are expression evaluations, which are only strings evaluated by the compiler and don't have an associated Ring object.

[6] `https://www.eclipse.org/m2e/`

In this paper we defined a simple criteria to evaluate four different code-history trackers. We also present a solution to model dependencies between changes that makes Epicea feature-complete from the point of view of the aforementioned criteria.

References

[DAB+11] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an interexchange format and source code model family. Technical report, RMod – INRIA Lille-Nord Europe, 2011.

[DCD13] Martín Dias, Damien Cassou, and Stéphane Ducasse. Representing code history with development environment events. In *IWST'13: International Workshop on Smalltalk Technologies 2013*, 2013.

[EVC+07] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo D'Hondt. Change-oriented software engineering. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference*, ICDL '07, pages 3–24. ACM, 2007.

[Gol84] Adele Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984.

[RL07] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166:93–109, January 2007.

[STCH12] Bastian Steinert, Marcel Taeumel, Damien Cassou, and Robert Hirschfeld. Adopting design practices for programming. In *Design Thinking Research*. Springer, 2012.

[UG12] Verónica Uquillas Gómez. *Supporting Integration Activities in Object-Oriented Applications*. PhD thesis, Vrije Universiteit Brussel - Belgium & Université Lille 1 - France, October 2012.

[UGDD12] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Ring: a unifying meta-model and infrastructure for Smalltalk source code analysis tools. *Journal of Computer Languages, Systems and Structures*, 38(1):44–60, April 2012.