Supporting Simultaneous Versions for Software Evolution Assessment

Jannik Laval[a], Simon Denier[a], Stéphane Ducasse[a], Jean-Rémy Falleri[a]

[a]RMoD Team, INRIA Lille - Nord Europe, France, http://rmod.lille.inria.fr

Abstract

When reengineering software systems, maintainers should be able to assess and compare *multiple* change scenarios for a given goal, so as to choose the most pertinent one. Because they implicitly consider one single working copy, revision control systems do not scale up well to perform *simultaneous* analyses of *multiple* versions of systems. We designed Orion, an interactive prototyping tool for reengineering, to simulate changes and compare their impact on multiple versions of software source code models. We deal with requirements such as: interactive simulation of changes, reuse of existing assessment tools, and ability to hold multiple and branching versions simultaneously in memory. Specifically, we devise an infrastructure which optimizes memory usage of multiple versions for large models. This infrastructure uses an extension of the FAMIX source code meta-model but it is not limited to source code analysis tools since it can be applied to models in general. In this paper, we validate our approach by running benchmarks on memory usage and computation time of model queries on large models. Our benchmarks show that the Orion approach scales up well in terms of memory usage, while the current implementation needs optimization to lower computation time. We also report on two large case studies on which we have started to work with Orion.

1. Introduction

Software architecture evolution, change impact analysis, software quality prediction, remodularization are important tasks in a reengineering process [BA96]. They often require developers to make choices about future system structure such as changing the dependencies between packages. While software maintainers would greatly benefit from the possibility to *assess different choices*, in practice they mostly rely on experience or intuition because of the lack of approaches providing comparison between possible variations of a change. Software reengineers do not have the possibility to easily *apply analyses on different version branches of a system and compare them* to pick up the most adequate changes.

In an ideal world, it should be possible to compare multiple futures of a system in presence of different changes and choices. In this setting, a typical reengineering session would look like the following: the reengineer launches a visualization to get an overview and detect problems, performs analyses (assessing quality, running cluster analyses to get a better organization, computing software metrics, . . .), simulates some changes, and iterates this whole process a couple of times. After each step, the reengineer reruns his tools (visualization, analyses, . . .) on the original model or any *possible future* versions to assess whether the proposed changes have a positive impact on the system. When one change does not offer the expected added value or bring unexpected drawback, the reengineer rollbacks to a previous version and starts over with new changes. He can also compare different alternatives using various indicators (quality model, software metrics, software visualization) and finally decide which among all of these possible futures is the one he settles on [MT04].

What we can see from this scenario is that there is a need to (1) *navigate* into *multiple, alternative* futures of the same system, (2) *apply various analyses on such futures*, and (3) *compare results* of such analyses. Since we want to be able to manipulate different futures of the same system *simultaneously*, directly changing source code to build each alternative does not scale up. There is a need for a software model of the source code that allows a reengineer to perform the operations mentioned above.

Usually reengineering tools use an internal representation of the source code (AST for refactoring engine, simpler source code meta-models for others) [CI90]. Similarly our approach is based on a source code model on which source code changes are applied interactively. We present an approach which allows reengineers (1) to create several futures by performing changes and (2) compare them. Each future is a full-fledged model which can be assessed through

usual software metrics, quality models, visualization... Of course versioning systems have supported branching for decades. We propose to be able to navigate and apply changes to possible futures or branches without actually committing them in a code repository. Specifically, we propose an infrastructure which optimizes memory usage of multiple versions for large models, enabling to work interactively on multiple models. Moreover, the concepts supporting our infrastructure are generic enough to blend in many meta-models. Existing tools can be reused on top of such versioned models without adaptation.

In this paper we raise the problem of the scalability of such multiple futures and branching versions: is the scenario described above possible and practical on large systems? This question can be declined at the implementation level and at the user level. First, what do reengineers expect during the workflow, what tools do they need and what kind of feedback should tools provide? Second, what is the infrastructure to put in place to support it efficiently? How to support model manipulations (edition, analyses) of large source code models with many small modifications (class changes, method changes)? A naive implementation is to make a copy of the original model for each future version and to modify the copies. However, with this naive approach a lot of memory is wasted by copying unchanged model entities. For example, modifying one package in a system with 100 packages would imply 99 useless copies. A first theoretical analysis has been realized in [LDDK09] where different approaches are compared with respect to space and time costs.

This paper presents in Section 2 our vision for reengineering. Section 3 details the principles and challenges of the model-based infrastructure supporting our approach and Section 4 gives code samples of the critical parts. Section 5 provides benchmarks about the scalability of the model compared to a naive full copy approach as well as brief reports about two large case studies. In Section 6, we discuss how our vision could be done (less efficiently) with revision control systems. Section 7 presents related work and Section 8 concludes this paper.

2. Orion vision for reengineering support

In this section we present the vision behind this work. We motivate it with a scenario and draw requirements for the implementation of such a vision.

2.1. Efficiency in reengineering

A reengineer has basically four forces driving his work. He should: (1) identify issues, (2) solve issues, (3) avoid regression of the system, and (4) minimize costs of change [BA96]. While the first three items are checked externally (bug report, review, tests), the reengineer has larger latitude to assess changes and their cost. Often there are multiple solutions to solve an issue, and assessing the most adequate one is a challenge of its own. The reengineer usually relies on his experience and intuition to select the most promising candidate.

2.2. Motivating scenario

We describe now a scenario dealing with reengineering package dependencies, especially the removal of cyclic dependencies between packages. From the scenario, we extract general requirements for the Orion approach *i.e.,* the simultaneous analysis and comparison of multiple versions of the system and illustrate them with examples.

A relevant scenario. Our experience with identification and removal of cycles in large software systems [LDDB09] shows us that one of the key challenges is to eliminate a cycle without creating a new one. Let us take an example from Moose, a platform for software analysis and reverse engineering [NDG05] (See Figure 1.a). In the original model, we are interested in three packages, two classes, and the two methods Model::inferNamespaceParents and Model::allNamespaces[1]. The black arrow from inferNamespaceParents indicates a reference to class Namespace. The gray arrow from inferNamespaceParents indicates an invocation of method allNamespaces. They create a dependency from package Moose-Core to respectively package Famix-Core and package Famix-Extensions. The dotted arrows from Famix-Core to Moose-Core and from Famix-Extensions to Famix-Core indicate dependencies of the same kinds seen at package level (coming from classes not shown in the figure). Altogether, the three packages make

[1]Notice that *allNamespaces* is defined in a different package than its parent class. This feature called *class extension* (or *partial class*) is especially useful to make packages more modular.

a strongly connected component. This component can be decomposed into two circuits: Moose-Core depends on Famix-Core and reciprocally, but Moose-Core also depends on Famix-Extensions, which depends on Famix-Core, which comes back to Moose-Core.

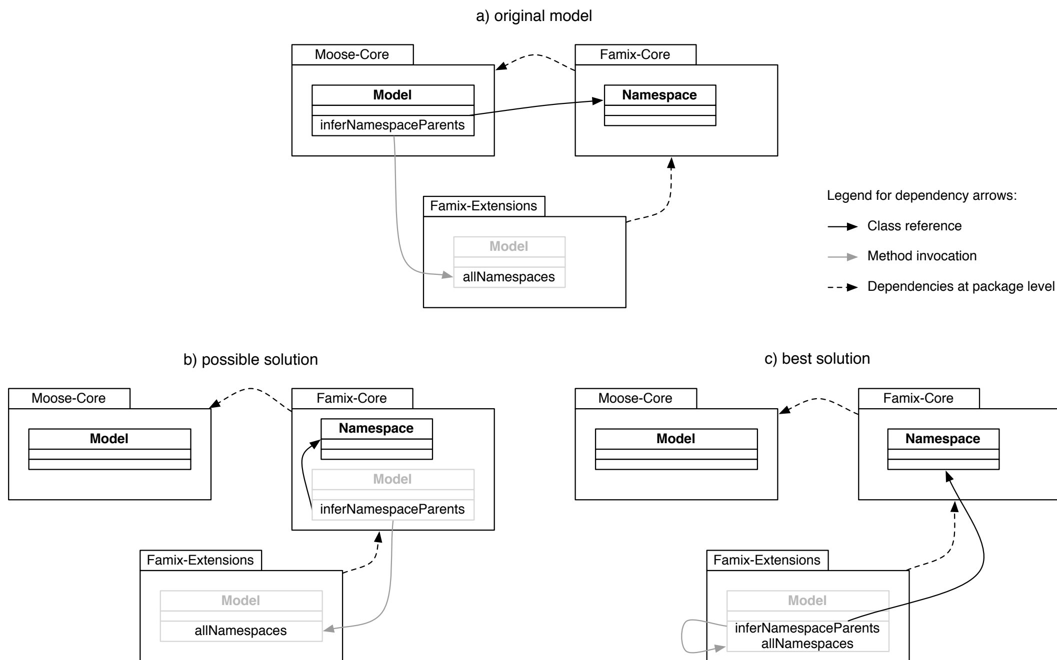

Figure 1: a) two circuits between three packages; b) a change that removes two circuits but creates a new one; c) a change that effectively removes two circuits.

Possible changes. In Figure 1.b, inferNamespaceParents, which is directly involved in one cycle, is changed into a class extension in package Famix-Core. As a consequence both previous cycles are broken since there is no dependency coming out of Moose-Core. The reference to Namespace is now internal to Famix-Core. However, the invocation escapes the package and a new cycle is actually created between Famix-Core and Famix-Extensions. Overall, this solution is possible but not good, because the new cycle is a regression.

In Figure 1.c, inferNamespaceParents is changed into a class extension in package Famix-Extensions. Now the invocation is internal to Famix-Extensions, while the reference escapes the package but "blends" into the existing dependency from Famix-Extensions to Famix-Core. No new dependency is created at package level, while the two previous circuits are effectively removed. In this case, a *single* cheap change cuts two circuits, which is a very positive outcome.

2.3. Requirements

From the preceding scenario, we extract a list of requirements. In general, removing cycles is hard because predicting the full impact of a change is difficult, be it positive or negative as illustrated in the above example. From this experience, we see that having the possibility to compare two solutions applied to the same original code model would help reach a decision. Taking such scenario as an illustration, we extract the following requirements for an infrastructure supporting this vision.

The reengineer needs access to different tools to assess the current situation: system structure (as a diagram or other visualizations), algorithms, metrics, queries to compute relationships between entities of the system. For our example, one needs graph algorithms such as Tarjan [Tar72] to compute cycles between packages. This also implies running queries over the entities of the model to build the graph of dependencies between packages. In [LDDB09], we develop a dedicated visualization using Dependency Structural Matrix to analyze in details cyclic dependencies

between packages. Metrics such as the number of cycles (as strongly connected components in the graph) are also useful to provide a quick assessment of system status.

> [Req1] *The reengineer needs specific tools to assess the current situation.*

The reengineer needs change actions to derive new versions of the system. Such actions have to be at the appropriate level of granularity for the task at hand. For example, changing package dependencies can involve many actions at different levels: moving classes around, moving methods between classes, merging/splitting packages. Tackling the problem of cohesion and coupling in classes requires more fine-grained actions to split methods and move attributes.

> [Req2] *Reengineering actions must have suitable outcome and granularity for the task at hand and need for a catalog of such actions.*

The reengineer needs to run the same set of tools on the original model and on derived versions to analyze the new systems and assess whether he reached his goals. He then can decide to stop and select this version, continue working on this version, mark it as a "landmark" and derive from it a new version to work on, or come back to another version and starts in a new direction. For example, an often unforeseen yet common consequence of cycle removal is the creation of a new cycle in another place (see Figure 1.b). At this point, the developer has two possibilities: he continues to work on this version to also remove the new cycle; or, he considers this new cycle too costly to fix and comes back to a previous version to work out a different solution.

> [Req3] *Tools should run indifferently on the derived versions as on the original model.*
>
> [Req4] *The reengineer can navigate between versions and start branches wherever he needs.*

The reengineer needs to assess what changed between two versions, to follow the impact of a change on the system and the progression towards a goal. This involves the same assessments as in *[Req1]* with a focus on change: changed entities directly impacted by the actions, but also changed properties of the system, or difference between two measures of a metric. He may eventually design custom tools, such as dedicated visualizations, to look at changes from the point of view of his task. For example, after performing a change, the reengineer should be informed of the destruction and creation of cycles. He can follow his overall progression by looking at the total number of cycles for each version.

> [Req5] *The reengineer needs custom tools to assess changes between versions.*

Finally, the reengineer settles on a version and wants to create this version starting from the original model. He needs the sequence of actions to apply, derived from the branch of the selected version.

> [Req6] *The reengineer needs a way to select and apply the appropriate changes.*

[Req1, Req2, Req5] stress the fact that the reengineer needs specific tools appropriate for the task at hand. Developing tools is costly, thus being able to *reuse existing tools* is an important asset for any reengineering infrastructure. On the other hand, *[Req3, Req4, Req6]* do not depend on the kind of task, and define the requirements for a generic reengineering infrastructure. In the following subsection, we present how the Orion approach embodies the above requirements. This presentation shows how specific requirements for the task of cycle removal and generic requirements for reengineering interplay in the front-end user interface. The remainder of the paper is dedicated to a more in-depth review of how Orion manages the generic requirements.

3. Orion design and dynamics

This section presents the design and challenges for the realization of Orion requirements. We first present the meta-model of Orion as well as its efficient implementation using shared entities: to save memory space and creation time, entities which do not change are shared between different versions of the model. We explain the creation of a new version and the dynamics of actions on a version. Finally, we detail how queries are resolved in the context of versions and shared entities. In particular, we show that a model should be *navigated from one specific* version even if a query may navigate to shared elements that are reused from older versions. Note that the infrastructure we present is not specific to source code meta-model but can be applied to any meta-model.

3.1. Orion meta-model: core and FAMIX integration

Orion core meta-model. The Orion approach is built around the three main elements shown in bold in Figure 2 (OrionModel, OrionEntity, and OrionAction). One instance of OrionModel stands for one version of the system. Each version points to its parentVersion, building a tree-like history of the system. The tree root represents the original model and contains all entities from the current source code. Hence, a version derives from a single parent but can have multiple children as concurrent versions are explored. Each OrionModel owns its OrionEntities. The system also contains a single OrionContext, which points to the current version on which the reengineer is working. Thus, navigating between versions is as easy as changing the OrionContext to point to the wanted version *[Req4]*.

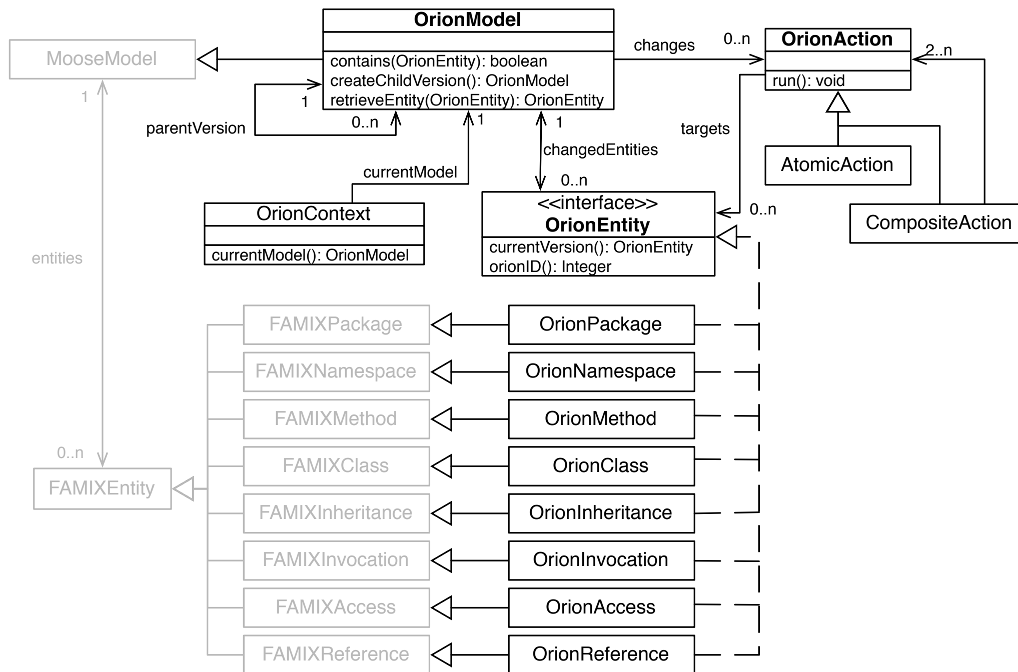

Figure 2: Orion meta-model

An OrionEntity represents a structural entity or a reified association between entities in the model. Orion entities represent the level of abstraction upon which reengineering actions need to be performed. For the task reported in this paper, we support four kinds of entities: OrionClass, OrionMethod, OrionPackage, OrionNamespace, and four kinds of association: OrionReference (from one class to another), OrionInvocation (of method), OrionInheritance, and OrionAccess (from a method to a variable).

Each OrionEntity has an orionID which is unique across all versions. A newly created entity receives a new, unique orionID. A changed entity keeps the same orionID as its ancestor. This identifier allows Orion to keep track of changed entities between different versions of the system.

OrionAction is the superclass for different kinds of actions. We distinguish between AtomicActions such as "remove a method", "move a class", or "create a package", and CompositeActions such as "merge two packages" or "split a class", built using the composite pattern. An instance of OrionAction runs to modify the current version but also stores information about the execution of an action (current version, target entity, specific parameters of the action) to keep track of changes. When executed, an action runs on the current model in OrionContext and modifies the entities in place.

FAMIX meta-model integration. Orion is an extension of FAMIX, a family of meta-models which are customized for various aspects of code representation (static, dynamic, history). FAMIX core describes the static structure of software

systems, particularly object-oriented software systems[2]. Extending FAMIX is a major asset of Orion as it allows us to reuse tools and analysis developed on top of FAMIX [DGKR09]. Especially, it fulfills requirement *[Req3]* which states that *tools should run indifferently on the derived versions as on the original model*.

In practice, the original model is created as a regular FAMIX model before being imported into Orion. During the import, FAMIX entities upon which actions can be applied are converted to their corresponding Orion entities. Other FAMIX entities which currently do not support OrionAction are directly included in the Orion model (for example, FamixVariable and FamixParameter). An Orion model deals seamlessly with both FAMIX entities and Orion entities.

3.2. The need for sharing entities between versions

Models for reengineering are typically large because they reify lots of information to perform meaningful analyse. For example, one system under study with Orion is Pharo[3], an open-source Smalltalk platform comprising 1800 classes in 150 packages. Its FAMIX representation counts more than $800,000$ entities, because it includes entities for variables, accesses, invocations... It becomes a major concern for an approach such as Orion, because we need several such models in memory to enable an interactive experience for the reengineer. In the following, we briefly review some of the strategies we analyzed in [LDDK09] and explain the dynamics of our model with shared entities.

The most straightforward strategy is the full copy, where a version is created by copying all entities from its parent version. Then two versions are two independent models in memory and tools run as-is on each model. However, this approach has a prohibitive cost both in term of memory space and creation time. In early experiments analyzing Pharo, each model took 350Mo in memory and, more annoyingly, copy took more than one hour to allocate and create the $800,000$ instances for each version. This was useless in the context of our approach.

Another common strategy is the partial copy approach. The principle is to copy only the entity changed as well as the entities connected to it, so that they still make a consistent graph in the current version. Unfortunately, this view does not hold in the FAMIX meta-model where all entities are transitively connected together through their relationships (each class representation points to its methods while each method representation points to its parent class). Thus, copying an entity and its linked partners comes back to copying the full model.

Our solution is a simplification of the partial copy approach, but requires an adaptation of the access of entities through links. The trade-off is between the memory cost of large models and the time cost of running queries on such models. Only entities which are directly changed are copied (then modified) in our approach. Other entities are left unchanged in their version, making the copy "sparse" and efficient. Changed and unchanged entities are reachable from the current version through a reference table, which is copied from the parent when creating the new version and modified by actions. Thus entities are effectively shared across different versions. However, dynamics are more complex than a simple Copy-on-Write standard approach as explained below.

Figure 3 illustrates how an Orion system manages the three kinds of change for an entity: creation, change, and deletion. Figure 3.a shows the original model with four OrionEntities with orionId **1** to **4**; the light gray area on the left represents the reference table, which holds pointers to each OrionEntity. Figure 3.b shows a child version where entity **5** has been created; the reference table holds a new pointer to entity **5** at the end (gray rectangle). Entities **1** through **4** are still accessible from the reference table. Figure 3.c shows another version where entity **4** has changed. Consequently, a new entity **4** appears in the version and the reference in the table is replaced with the newer pointer. Figure 3.d shows a version where entity **3** has been deleted. Only the pointer to entity **3** is really removed from the reference table, making the entity unreachable and effectively deleted from this version.

Table 1 summarizes the pros and cons of the above approach for our constraints. Numbers in parenthesis refer to the Pharo case study, which is our largest case to date with $800,000$ entities per model.

3.3. Running queries in presence of shared entities

Queries are the foundations for analysis tools as they enable navigation between entities of the model. Basic queries represent direct relationships between entities: a class can be queried for its methods, a method can be queried for its outgoing invocations (*i.e.,* method calls within the method), a package for its classes, ... More complex queries made by analysis tools are composed from such queries.

[2] see [DTD01] and http://www.moosetechnology.org/docs/famix

[3] http://www.pharo-project.org/home

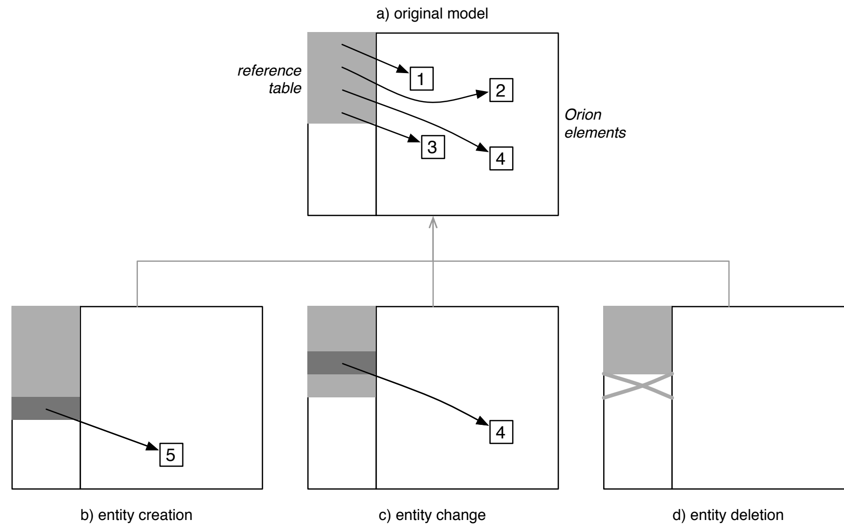

Figure 3: illustration of Orion dynamics through different changes (creation, change, deletion)

Approach	creation cost for new version	memory cost for x versions	access cost to an entity
Full copy in FAMIX	copying all entities (70 minutes)	x * original size (350 Mo per version)	direct access
Scarce copy + table look-up in Orion	copying the reference table (30 seconds)	original model size + x times reference tables + size of each changed entity (350Mo + around 10Mo per version)	table look-up

Table 1: comparison between copy model and shared entity model in the case of Pharo

Sharing entities across different models have an important impact on the way queries are run in a version. In particular, starting from a given version, a query may run on shared entities from older versions: results returned by such shared entities must always be interpreted in the context of the starting version, as older entities may link to entities which have changed since. This specific aspect makes our solution more subtle to implement than a simple Copy-on-Write. The challenge of running queries over shared entities is summarized as follows:

1. basic queries retrieve entities which may or may not reside in a parent version;
2. then Orion should resolve each retrieved entity to its most recent entity (sharing the same orionId) reachable from the current version.

The challenge is akin to late binding in object-oriented languages. An entity residing in a parent version is always interpreted in the context of the current version where the query is run, same as a method invocation is always resolved against the dynamic class of this, even when the call comes from a method in a superclass. In our solution, there is no look-up through parent versions to resolve the most recent entity, but a direct access through the reference table of the current version.

Example: some changes. Let us illustrate this challenge with the following case. In Figure 4, two changes are applied on the original model v1 (top diagram): a class deletion (class Student is removed in version v2) and a method move between classes (method speak() moves from Professor to Person).

The deletion action directly impacts the parent package Actor. In the new version v2, class Student is removed and package Actor should be updated so that it does not reference Student. First, v2.Actor is created as a new OrionEntity (with the same orionID as v1.Actor) as it only knows about Person and Professor; second, Student is not in the reference table of v2 (left sidebar) so it is unreachable.

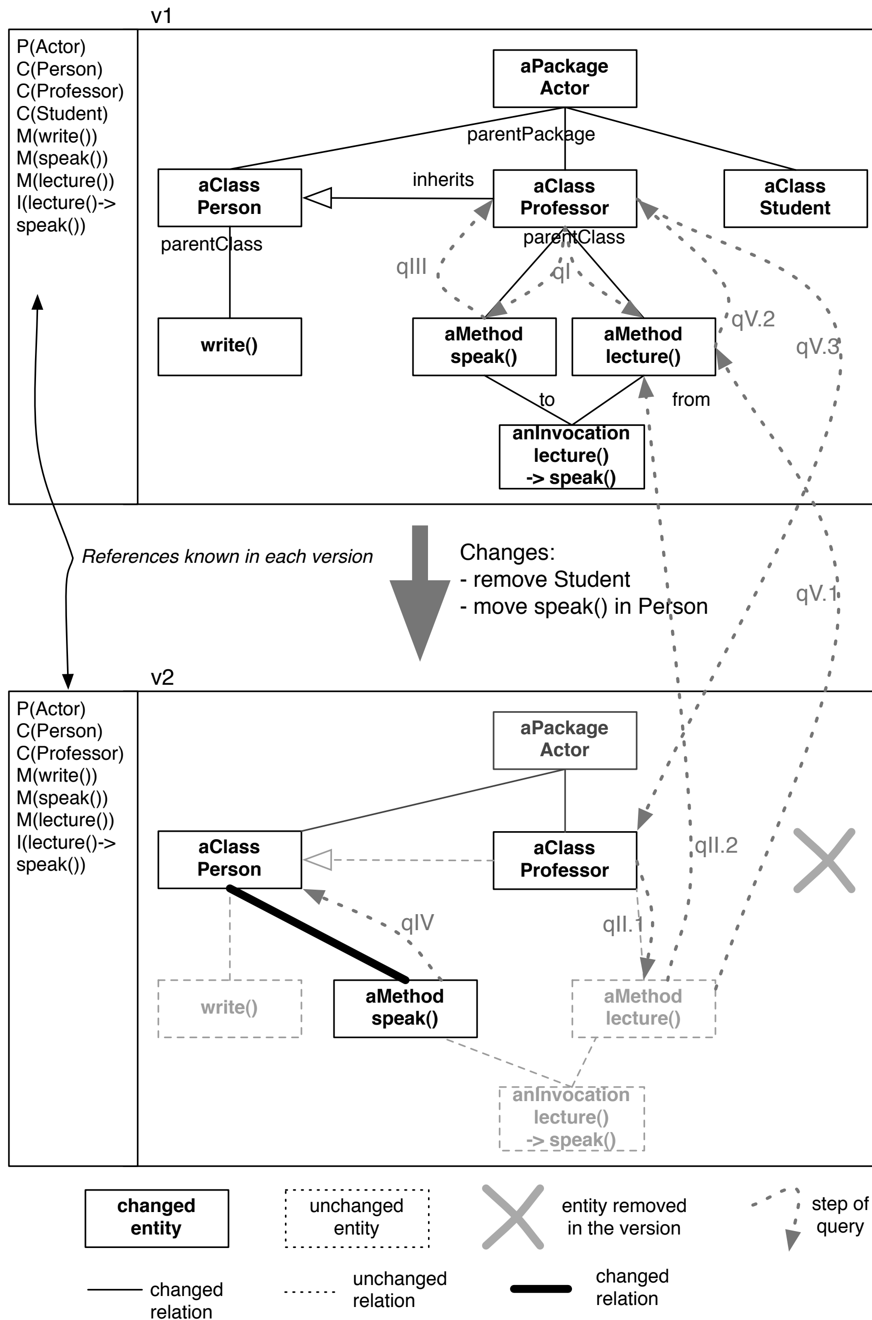

Figure 4: sample model with one derived version: classC is deleted from Actor and method mB1() moved from classB to classA

The second change involves three OrionEntities for which new versions are created to mirror changes: v2.Professor does not contain method speak() anymore while v2.Person now contains it, and v2.speak() itself now refers to v2.Person as its parent class. Notice that the invocation lecture()→speak() is not touched by this change as it is still considered as an invocation on method speak(). Methods lecture() and write() are not updated in v2 because they

are not directly impacted by the changes from v1 to v2.

Notice how we use the dotted notation version.element to refer unambiguously to an OrionEntity residing in a version. For example in Figure 4, v1.Person refers to Person in the original model. v2.Person is a new element which shares the same orionID. v1.write() and v2.write() represent the method write() in their respective version, but the entity is actually shared.

Example: some queries. The following queries illustrate, from basic to more challenging cases, how navigation across shared entities is resolved in Orion. A query takes two parameters: a target entity and the current version as a context (v1 or v2). The general algorithm for processing basic queries takes two steps. First, the query is actually run against the target entity and returns entities which possibly originate from different versions. Second, for such entities, orionIDs are matched against the reference table of the current version to retrieve the latest entities corresponding to each orionID.

In Figure 4, several queries are represented, from basic to more complex, which we explain below. qV and qVI especially illustrate the challenge of shared entities.

- qI – v1.Professor.getAllMethods() → {v1.speak(), v1.lecture()}. This query returns all methods of the class Professor in the context v1.
- qII – v2.Professor.getAllMethods() → {v1.lecture()}. lecture() exists in v2 but resides in v1, because this entity has not been modified. This is the standard case of shared entities between versions. Since lecture() is in the reference table of v2, it is reachable. Since a specific v2.lecture() does not exist, the reference actually points to the most recent entity with respect to v2, which is v1.lecture().
- qIII – v1.speak().getParentClass() → v1.Professor. This query runs in the context of v1, giving the original view of the model.
- qIV – v2.speak().getParentClass() → v2.Person. This query runs in the context of v2. It returns a different result than qIII respecting changes applied in v2.
- qV – v2.lecture().getParentClass() → v2.Professor. As noted for qII, v2.lecture() is actually v1.lecture() and the query is run against the later (represented by step qV.1 in Figure 4). Then the query resolves in two steps: first message parentClass sent on v1.lecture() retrieves v1.Professor; second, since the query runs in the context of v2, Orion retrieves the correct v2.Professor from the reference table of v2 using the orionID as the common denominator.
- qVI – v2.lecture().getParentClass().getAllMethods() → {v2.lecture()}. This is a composed query. Here we get the same scenario than with qV (v2.Professor) but in addition we query all its methods as in qII, which returns its sole method v1.lecture() in the version v2.

Queries qV and qVI show the subtlety of running queries on shared entities. First, query qV is launched on v2.lecture() but actually runs on v1.lecture(). Second, when a query is run in a version, it must follow changes related to this version: in query qVI, v2.lecture() is v1.lecture(), but querying element Professor selects entity v2.Professor, not v1.Professor, so that the correct set of methods is retrieved. The last query stresses that even if an entity from a parent version is returned, traversing this entity implies a resolution against the current version to retrieve changed entities. In presence of shared entities between versions, a composed query may reach entities of a parent version, not residing in the current version, yet it should always return entities as seen from the current version. A version acts as a view on a graph and from such a view the graph and its navigation should be consistent.

4. Implementation

4.1. Core implementation

We detail some spots of the implementation which illustrate the dynamics of Orion. Our goal is to give enough information so that the approach can be reproduced in other meta-modeling environments such as the ones supporting EMF. We give code samples in pseudo-code for the following cases: creation of a new version, action execution, basic query. It shows Orion internals, which create changed entities and resolve an entity in the current version.

Version creation. A new version is created from a parent version by copying the full list of references from the parent (only references are copied, not entities). The version also stores a reference to its parent version and the parent adds the version as a new child:

```
OrionModel::createChildVersion(): OrionModel {
    OrionModel newVersion = new OrionModel();
    childrenVersions.add(this);
    newVersion.setParentVersion(this);
    for(OrionEntity entity: entities()){
        newVersion.addEntity(entity); }
    return newVersion; }
```

Action execution (Move method). An instance of OrionAction runs to modify the current version but also stores information about the execution (current version, target entity, specific parameters of the action) to keep track of changes. Parameters are stored in instance variables of the action, set at initialization. To move a method from its current class to another class, ActionMoveMethod needs three parameters: the current version as orionContext, the method as targetEntity, and the target class as targetClass.

The method run of the action retrieves the entities concerned by the change from the orionContext then directly update these entities. The orionContext (the current version) takes care of copying entities from the parent version and updating its reference table. ActionMoveMethod touches three entities: the method and the two classes. Its method run updates the links between those three entities to apply the change.

```
ActionMoveMethod::run(): void {
    OrionMethod method = orionContext().retrieveEntity( targetEntity() );
    OrionClass oldClass = orionContext().retrieveEntity( method().parentClass() );
    OrionClass newClass = orionContext().retrieveEntity( targetClass() );

    oldClass.methods().remove(method);
    newClass.methods().add(method);
    method.setParentClass(newClass); }
```

The method retrieveEntity from OrionModel first checks whether the entity resides in the model. In this case it means that the entity has already been changed and can be directly modified. Otherwise, it makes a shallowCopy of the entity since it comes from a parent version. shallowCopy copies only references to other entities as well as the orionId.

```
OrionModel::retrieveEntity(OrionEntity anEntity): OrionEntity {
    if( contains(anEntity) ) {
        return anEntity;
    } else {
        OrionEntity changedEntity = anEntity.shallowCopy();
        changedEntity.setModel(this);
        return entities().add(changedEntity);
    }}

OrionModel::contains(OrionEntity anEntity): boolean {
    return this == anEntity.model(); }
```

Query execution. The main concern of queries in Orion is that they always return entities as seen through the current version. Basic queries, which directly access entities in a model, needs to be adapted in Orion to resolve the direct access in the context of the current version.

```
OrionMethod::parentClass(): OrionClass {
    return parentClass.currentVersion(); }
```

The naive implementation of currentVersion below looks for the entity with the same orionId in the current model. However, it involves the traversal of the reference table (entities()) each time an entity needs to be resolved. Care is needed to optimize this method as well as the traversal. A straightforward optimization is to first test whether the entity belongs to the current version using OrionModel::contains (see above), otherwise to launch the traversal. A more general optimization is to use an efficient data structure such as a search tree to implement the reference table.

```
OrionEntity::currentVersion(): OrionEntity {
    for(OrionEntity entity: OrionContext.currentModel().entities()){
        if( entity.orionId == orionId ) {
            return entity; } }
    return null; // should never happen
  }
```

Complex queries are built on basic queries to compute more information. They always get the most recent entities from basic queries and thus do not require adaptation in Orion. This is especially interesting as most analysis are built with complex queries, enabling reuse of existing tools, while basic queries (which require adaptation) form a limited set fixed by the meta-model.

4.2. Experimental Orion browser: removal of cyclic dependencies between packages

We designed and implemented Orion, an infrastructure for reengineering, as a realization of the requirements of Section 2 and proof of concept of our vision. We developed an experimental version browser dedicated to analysis of cyclic dependencies between packages. It serves as the central place for running version analyses and cycle analysis (shown in Figure 5). This section describes the core functions of the browser with respect to our vision, illustrated in the context of cycle removal. We do not claim that this browser is the sole solution to manage simultaneous versions. It is an illustration of our vision and of possibilities currently offered by Orion.

We distinguish two parts in the browser layout: top row for navigation in the system, bottom row for task analysis and change assessment. The reengineer interacts with the browser by selecting elements in the different panels and opening a contextual menu.

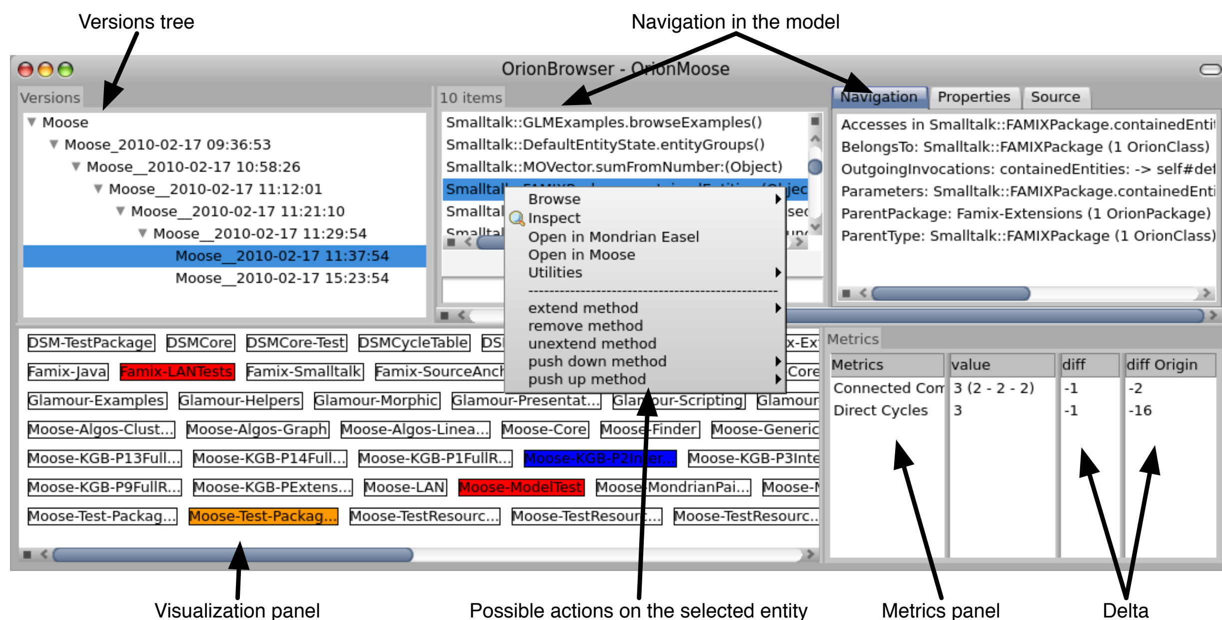

Figure 5: an Orion browser dedicated to cycle removal

The navigation row at the top is built from two main panels: the left-most panel shows the tree of model versions. The second panel is a sliding navigation browser of the selected version, embedded in the Orion browser (middle and right panels in Figure 5). It first displays the list of entities by group (all packages, all classes…). It is possible to browse entities by selecting a group. Then by selecting an entity, a new panel opens on the right of the current panel with all linked entities (in Figure 5, selected method in middle panel displays groups of linked entities in right panel (accesses, invocations…), which can be browsed in turn). In the first panel (left), one can create a new child version

from the selected version, or delete an existing one (from a contextual menu) *[Req4]*. Another action accessible on the selected version is to display the sequence of actions leading from the original model to the version *[Req6]*. Each entity in the second panel (middle) defines the list of reengineering actions enabled on itself, like moving a method to a class, or moving a class to a package *[Req2]*. A dedicated contextual menu is accessible, listing possible actions (see the pop-up menu in Figure 5). Applying an action produces a change which is recorded in the currently selected version. By default, the reengineer can browse and filter the full set of entities in the model. But the reengineer can also switch to changed entities in the second panel to only show entities which have already changed in the selected version.

The bottom row is for analysis and assessment. It contains one large visualization panel and another panel for displaying metric measures. This part of the browser is customized with specific visualizations and metrics for the current task *[Req1, Req5]*. It shall not necessarily display all information relevant for the task at hand, but rather be a starting point to launch more complex and intensive analysis, depending on the assessment of the reengineer for the current situation *[Req3]*.

Visualization. Figure 5 shows a very simple visualization in the bottom left panel dedicated to highlight strongly connected components (SCC) involving packages. This visualization has been chosen because it is both space-savvy (not requiring a complex graph layout nor a large matrix) and time-efficient (using Tarjan algorithm for detecting SCCs in linear time [Tar72]). Each boxed label represents one package. Colorized boxes indicate packages which belong to the SCC involving all packages of the same color. Hence, the reengineer gets a fast overview of cyclic dependencies in the system and can focus on each problematic subset (that is, each SCC) separately. From the visualization, he can select a single SCC and launch from a contextual menu a more sophisticated visualization such as eDSM [LDDB09] to perform a detailed analysis and devise the plan of actions (not shown).

Change impact metrics. The bottom right panel displays properties of the system dedicated to the task at hand. Such properties are chosen to assess the current version with respect to the global goal and to follow the progression from the original model. In line with the visualization, we only use simple metrics in the browser. More sophisticated metrics are only useful in specific analysis. In this case, following the number and size of strongly connected components is a good indicator of progression toward the objective (no cycle implies no SCC). We also include the number of direct cycles, which are a primary target for reengineering. Figure 5 shows these two metrics with their values and two change indicators: difference with the parent version and difference with the original version. More specifically, the value of "Connected Components" is *3 (2 - 2 - 2)*, which indicates three strongly connected components, each composed of only two packages. This is of good omen for the next versions. The last column of "Direct Cycles" shows -16, which means that 16 direct cycles have been removed in this version, compared to the original model.

Again this browser is just an illustration of the Orion infrastructure. We present in the next section the model which implements the generic requirements of the reengineering infrastructure and allows this browser to work.

5. Benchmarks & case studies

In this section we run three benchmarks showing that Orion scales up well in terms of memory usage without slowing too much the time necessary to run queries. We compare our approach with the full model copy, which defines the baseline for computation time (no overhead at all) but does not scale up well in memory. The first benchmark shows that Orion saves a lot of memory compared to the full copy approach. The second benchmark shows that the time overhead induced by Orion on queries is acceptable for an interactive experience. The third benchmark shows that the creation time of a new version is insignificant in Orion, while it is fairly slow for the full copy approach, making it impractical in an interactive scenario. We also report on two case studies undertaken on large projects, with insights on the initial changes performed with Orion.

5.1. Test data

We ran our benchmarks against a model of Moose imported in FAMIX [NDG05]. We will call this model *Famix-Moose* from now on. Famix-Moose is a typically large model with more than 150,000 entities, including 721 classes in 69 packages, 8,574 methods, 65,378 method invocations, and several other entities.

5.2. *Memory benchmark*

Objective This benchmark shows the difference in memory usage between Orion and the regular Moose. We devise two settings to assess this benchmark: one with a few changes per version (low impact setting), and one with many changes per version (high impact setting). The goal of the two settings is to check how the memory usage of Orion is impacted by small and large changes.

Experimental settings

Regular Moose. In this experiment, we first create the Famix-Moose model, and we copy it 20 times. Between each copy, we measure the memory used by the different models.

Orion with low impact setting (Orion Low). In this experiment, we first create the Famix-Moose model. Using Orion, we create 20 successive versions (leading to a total number of models in memory of 21). For each version, we do the following operation 5 times: We randomly select two methods and add an invocation between them. This modification impacts only the two concerned methods, so at most 10 entities are changed in each version (provided each method is selected once per version). Between each version creation, we measure the memory used by the different models.

Orion with high impact setting (Orion High). In this experiment, we first create the Famix-Moose model. Using Orion, we create 20 successive versions. For each version, we do the following operation 10 times We randomly select a method and delete it. The deletion of a method has a high impact because it changes multiple entities: linked invocation entities are deleted, methods invoking or invoked are changed, as well as its parent class and parent package. For instance in the run of this experiment, methods such as **=** or **do:** have been removed, forcing the copy of respectively $2,917$ and 813 elements. Between each version creation, we measure the memory used by the different models.

Results Figure 6 shows benchmark results. First point represents memory usage of the infrastructure and of the original model, which takes up to 100 mega-bytes. It is clear that copying the full Moose model induces a huge memory usage. For instance, 10 such models require almost 600 mega-bytes of memory, which is a lot even for a recent computer. On the other hand, Orion behaves very well from this point of view. To store the 20 new versions with Orion, only 220 mega-bytes are sufficient for the low setting, and 230 mega-bytes for the high setting, which is a huge improvement. The difference between low and high settings is due to the change of many entities in high setting. It makes versions of multiple entities stored in the reference table. We can also see that Orion is robust even when the changes are complex.

5.3. *Query time benchmark*

Objective This benchmark shows the difference in query running time between Orion and the regular Moose. Using a full copy of a Moose model actually boils down to using the regular Moose system in the day-to-day usage, which defines the baseline for timing queries. We devise two settings to assess this benchmark: one with a few changes per version (low impact setting), and one with lots of changes per version (high impact setting). The goal of the two settings is to check how query performance of Orion is impacted by small and large changes.

Experimental settings

Regular Moose. In this experiment, we run the queries on a standard Famix-Moose model. Since copying the model does not affect the time spent in queries, we run the experiment on a single version. We perform 4 queries:

- invokedMethods, on each class of the model. This query returns all methods invoked by methods of all classes. We chose this query because it runs over all methods, browsing both changed and unchanged entities. This query is simple but retrieves a large result, while most analysis are performed on subsets with complex queries.
- allMethods, on each package of the model. This query returns all methods contained in all classes of all packages.

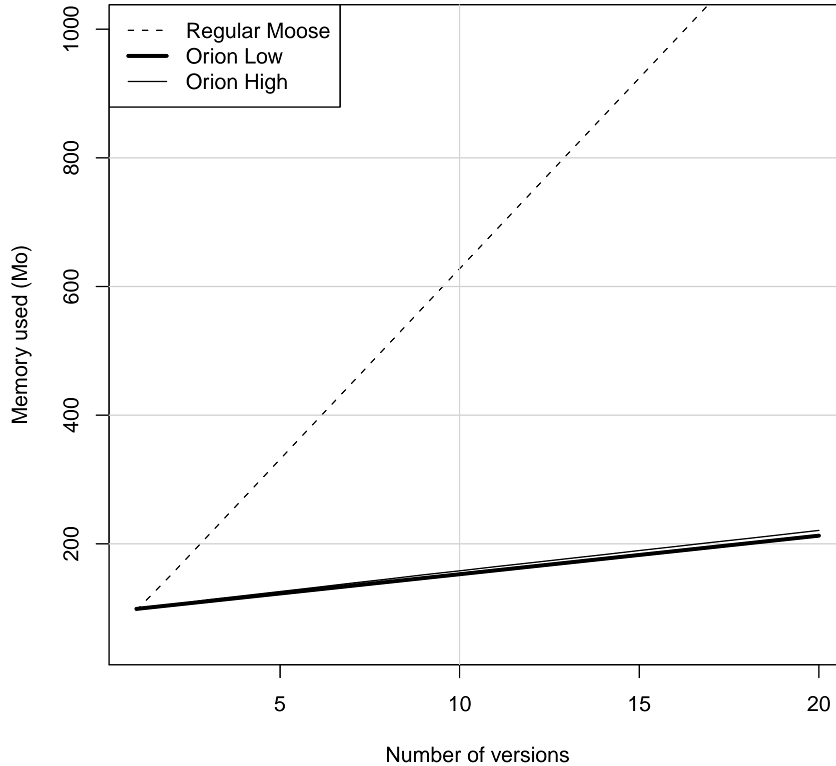

Figure 6: memory usage in Orion vs full copy. The x-axis shows the number of versions and the y-axis shows memory usage (in mega-bytes)

- allSubclasses on each class. This query is simpler than the two preceding, but it represents a good indicator for usual queries
- superclass on each class. This query return a simple result, as the previous one.

We perform these queries 10 times and take the mean execution time.

Low impact setting (Orion Low). In this experiment, we take the same setting as *Low impact setting* in Section 5.2. For each version, we measure the mean time spent to run one query. We perform the five queries on each class (or package, depending of the scope) of the model. We run these queries 10 times and take the mean time.

High impact setting (Orion High). In this experiment, we take the same setting as *High impact setting* in Section 5.2. For each version, we measure the mean time spent to run one query. We perform the five queries on each class (or package, depending of the scope) of the model. We run these queries 10 times and take the mean time.

Results Figure 7 shows the result of the query invokedMethods. It shows the worst result of the 4 queries. It shows also that results are linear during versions. Table 2 shows benchmark result averages. We give only average because results are linears. Orion induces an overhead on the time spent in queries which is acceptable in view of query time. This overhead does not depend on the number of versions nor on changes but on the structure used for the reference table. Nevertheless, the time overhead is not so important (see column Factor in Table 2) as to disturb the interactive experience. The High impact setting provides approximately same results than Low impact setting.

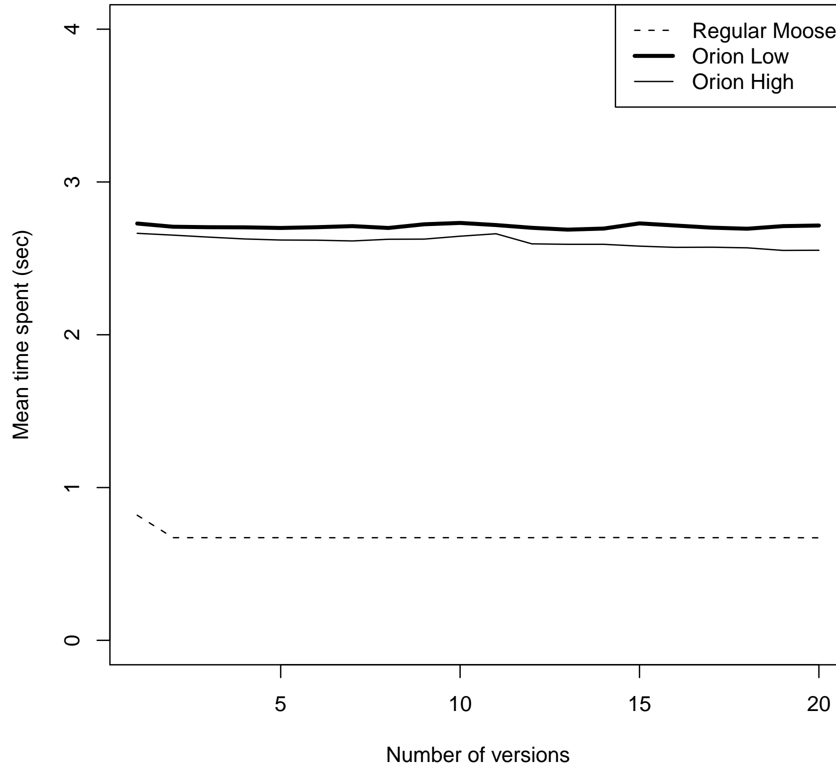

Figure 7: comparison of the time spent in queries of Orion vs Regular Moose. The x-axis shows the number of versions and the y-axis shows the time spent (in seconds)

	Regular Moose	Orion Low	Orion High	Factor
Invoked methods	735.25	2708.4	2653.45	3.64
All methods of each package	9.9	19.05	20.7	2
All subclasses of each class	3.45	6.65	6.6	1.92
Superclass of each class	4.55	5.4	5.45	1.20

Table 2: Average of Query time (in milliseconds)

5.4. Creation time benchmark

Objective This benchmark shows the difference in creation time for a new version between Orion and the regular Moose. Creation of a new version should not hamper the interactive experience in the infrastructure, supporting the workflow of the re-engineer. With the regular Moose, the creation of a new version is a deep copy of the model. With Orion, it is only a copy of the reference table.

Experimental design

Regular Moose. In this experiment, we first create the Famix-Moose model, and we copy it 10 times. We measure the time spent in copying each version and then compute the mean time of copy for version creation.

Orion. In this experiment, we first create the Famix-Moose model. We perform 10 successive version creations (each version being the parent of the next one). We measure the time spent in creating each version. Finally, we compute the mean time of version creation. This experiment does not need change execution because Orion just copies the reference table during creation.

Results

- *Regular Moose:* $67,45$ seconds
- *Orion:* $3,15$ seconds

We can clearly see that the copy time of a Moose model (more than one minute) is impractical for an interactive tool. On the other hand, the version creation time required by Orion (around three seconds) is acceptable and should not hamper the user experience.

5.5. Case studies

We report on two case studies performed on large projects with Orion: Moose and Pharo. Moose is a software analysis platform comprising 721 classes in 69 packages. Its model contains $150,000$ entities. Pharo is an open-source Smalltalk environment comprising 1800 classes in 150 packages. For this case study, we ran a smaller model containing $685,000$ entities (only packages in cycles with their embedded entities) instead of $800,000$.

The goal of the case studies was the removal of cyclic dependencies between packages as shown in Section 2. To customize the Orion browser for this task, we developed the dedicated visualization as well as goal metrics tracking cycle changes (Figure 5). It allows us to follow the evolution of changes, and focus on packages in cycles, for which we reuse the eDSM tool (enhanced Dependency Structural Matrix), an advanced visualization for detailed analysis of cyclic dependencies [LDDB09]. Each case study involved two experts in front of Orion. One was a reverse engineering expert who assessed cycles based on report from the browser and eDSM, while the other was an expert of the system under study who suggested changes based on the previous assessment. Once a change is applied in Orion, visualization and metrics are computed again on the changed version to follow progression or regression of the system, and a new assessment can begin. The process can be repeated until the goal is met.

Moose. Initial assessment of Moose showed 17 packages (among 69) dispatched in 5 strongly connected components, one involving 7 packages. 19 direct cycles between packages were detected. 7 versions were created and assessed to achieve the final objective, which removed 15 direct cycles. The 4 remaining cycles actually relates to test resources, that is are deliberate cycles created to test Moose tools. The proposed plan of actions touches 52 entities in the model with 22 actions. All were basic actions (3 class moves, 1 reference removal, 1 method delete, 17 method extensions). Among those 22 actions, 13 were readily integrated into Moose source code. The remaining 9 changes, participating in the same single direct cycle, are pending, as the involved concern might be completely refactored.

Pharo. The complexity of the Pharo case study is much larger as initial assessment showed 68 packages in a single strongly connected component, creating 360 direct cycles. We only report on a single 2-hours session, as the case is still ongoing. 11 versions were created during this session, impacting 110 entities and removing 36 direct cycles. 29 actions were executed, 3 of which were composite actions (merge packages, remove class). Such actions effectively extracted 5 packages from the original SCC. All actions have been integrated in the development release of Pharo.

5.6. Threats to Validity

5.6.1. Benchmarks

The aim of benchmarks is to compare our solution with a naive duplication of models. Two threats to validity are highlighted: one external validity, one construct validity. External validity claims that the generality of the results is justified [ESaSD08, PPV00]. The threat is the meta-model used by Orion. Construct validity claims that the construction and the measures are correct. The threat in this part is the randomize changes selected for benchmarks.

Meta-model. The meta-model provides a reification of dependencies between elements. So, a model based on Famix has a lot of elements due to this reification. A meta-model without this kind of behavior is smaller and could have different results. But not so, because the maximal shared entities method is better than copying all elements in the model.

Randomize changes. A change should impact more or less elements in the model, due to their relations with other elements in the model. For example, the deletion of the method "Collection.at()" impacts a lot of entities in the model. Some other methods can be deleted with no impact because they are not used (*i.e.,* dead code). We pay attention that in each benchmark, methods with a lot of relationships are removed. In the results given in this paper, methods as "Collection.add()" have been removed.

5.6.2. Case studies

The case study made on the Moose project is possibly biased because we represent a part of the maintainers of the project and we therefore know the system. But we have noticed that some unpredictable impacts can appear, so we think by our own experience that this infrastructure is useful.

The case study on Pharo has no real threats to validity. One of the author worked with a Pharo maintainer: the first one knows Orion, the second knows Pharo. The ideal process is to have only one person who manipulates the two tools.

6. Discussion: reengineering using revision control systems

Revision control systems (like CVS and SVN) have managed version branching for decades. They offer compact way sof storing delta between versions but reengineering environments or IDEs like Eclipse do not take advantage of such incremental storage. For Eclipse, there is only one version of the source code in memory when we perform a given refactoring.

Using code versioning to support our scenario boils down to (1) create one branch in the code repository for each possible solution, (2) effectively apply changes in the code, then (3) run the analysis tools on each branch. The developer would eventually need some tools to compare versions. Note that we are not concerned by textual differences between versions but software quality assessment based on visualization and metrics of the different versions.

Such a process is possible but costly, as one has to check out multiple working copies, set up the reengineering tools for each version, apply effective changes in the code. In addition it is cumbersome to navigate and compare the versions. It makes it impractical to test numerous small solutions. In practice, developers often can not support such costs: they give their best guess at what would be the adequate solution, apply it, and rollback if it reveals too problematic.

Overall, this process has two drawbacks:

- it consumes time and resources, as the developer has to switch between analysis and reengineering environments, work directly with the code, version its code so as to move forward and rollback between changes. Moreover, it breaks the flow of work while one has to deal with these multiple concerns.
- due to these costs, it is impractical to compare multiple alternative solutions as one should produce the code for each solution.

These drawbacks are not present in Orion due to a single environment for analysis and reengineering. Moreover, as Orion works on models, it does not change source code, and make available comparison of multiple alternative solutions. It allows us to import only one time the source code as model and working on version without changing the original model.

The scope of this work can be discussed: we work on models, but we could work on source code. As we explain in Section 2, the reengineer needs specific tools to assess the situation (Req1). These tools are available in reengineering environment as FAMIX. If we had chosen to work on source code, we would have had to import each new version in the reengineering environment, or to make a bridge between the source code and the model. So the decision we

took is to make the implementation in the reengineering environment, and in future works to compute the change to applied them on source code.

All requirement listed in Section 2 could be implemented in Orion. We show them partially in Section 4.2, but all these requirement can be implemented in Orion:

- Req1. In the reengineering environment provided with FAMIX, we can assess the current situation thanks to all the reengineering tools.
- Req2. The granularity is the same provided by FAMIX. All the dependencies are reified and there is a granularity as fine as temporary variable.
- Req3. Orion provides a shared system which allows us to run tools on each version indifferently.
- Req4. The navigation and creation of branches and versions is easily usable, as shown partially in Section 4.2.
- Req5. Orion make use of the Moose environment provided around FAMIX. Some tools provide faster scripts to create custom tools for Orion, then comparing versions is available as they are all in memory.
- Req6. Orion keeps in memory all changes applied to a model. For now, Orion provides a list of changes to do in source code. But we could automate them, as we explain before in this paragraph.

7. Related work

In relation with this work, we highlight three domains: versioning mechanism, change management, and change impact analysis. To the best of our knowledge, there is no approach which supports the navigation and manipulation of multiple versions simultaneously in memory.

7.1. Software configuration management & revision control

Software Configuration Management (SCM) is the discipline of managing the evolution of a software system. It integrates Revision control which is the management of changes. It is the predominant approach to save software evolution. It allows one to manage high-level abstraction evolution.

The majority of revision control systems use a diff based approach. they only store changes so they are efficient in memory. In our approach, we need a compromise between memory efficiency and a permanent graph access. So, the domain of revision control does not provide a model which allows us to navigate between multiple versions of a model. In fact, this is not a real goal of the revision control domain.

Smalltalk basic mechanisms to record changes dynamically is called a changeset. A changeset captures a list of elementary changes that can be manipulated, saved to files and replied if necessary. However, in Smalltalk systems only one single version of a system can be refactored at a time, even if changeset containing several versions can be manipulated. The same happens with Cheops and change-oriented methodology and IDES [Ebr09]. In [Rob08], the author argues that managing changes as first-class entities is better than traditional approaches. The implementation of this approach records fine-grained changes and provides a better comprehension of changes history. This approach is applied on a single version of source code. Orion integrates first-class changes as each action is represented by an object. It allows the developer to have fine-grained information.

In [BMZ+05], three tools are compared: Refactoring Browser, CVS and eLiza. A refactoring browser transforms source code. It has basic undo mechanism but does not manage versions. So, it is really useful for refactoring source code but it works on a current model of source code. It is not really adapted for the application of various analyses on different versions. CVS (Concurrent Versions System) works on file system and supports parallel changes. However since CVS does not include a domain model of the information contained in the files they manipulate, it is difficult to use a CVS model to perform multiple analyses on various versions. It is possible but limited. The third element compared in this paper is eLiza. This system from IBM has been created to provide systems that would adapt to changes in their operational environment. This system provides a sequential versioning system because only one configuration can be active. This system is not adapted to our subject because it is based on an automatic change system in relation with the environment.

Molhado [NMB05] is a SCM framework and infrastructure which provides the possibility to build version and SCM services for objects, as main SCM systems provide only versioning for files. As it is flexible, the authors work on several specific SCM built on Molhado: web-based application [NMT05], refactoring aware [DMJN07] to manage changes and merge branches. The main topic of Molhado is to provide a SCM system based on logical abstraction, without the concrete level of files management. This approach is orthogonal to Orion because it controls changes while Orion simulate changes. There are some similarities between the two approaches and it is probable that Orion could integrate a SCM, in the future.

7.2. Change impact analysis

Compared to Software Configuration Management (SCM) and Revision Control System, which supports change persistence and comparison, the domain of change impact analysis deals with computing (and often predicting) the effect of changes on a system. Our approach is orthogonal to change impact analysis. Tools performing change impact analysis can be used in the Orion infrastructure to perform change assessment on a version and guide the reengineer when creating new versions and testing new changes. We structure the domain in two parts: change model, which could inspire future work on the Orion meta-model; change assessment, which provides tools and analysis (metrics, visualization, change prediction).

Change model. Han [Han97] considers system components (variable, method, class) that will be impacted by a change. The approach is focussed on how the system reacts to a change. Links between these components are association (S), aggregation (G), inheritance (H), invocation (I). Change impact is computed based on the value of a boolean expression. For example a change is considered as S H+G. This work has been reused in [ALS06]. The class-based change impact model [CKKL02] is based on the same semantics, with a more general model. It analyzes history and identifies classes which are likely to change often. These approaches use the same type of link between elements as Orion. This type of analysis is not included in Orion, our approach provides metrics and visualization analysis based on the direct analysis of the model. In the future, it will be possible to integrate a similar approach, as Orion knows which entities are modified.

An history-based approach is Hismo [Gir05], a meta-model for software evolution. This approach is based on the notion of history as a sequence of versions. A version is a snapshot taken at a particular moment. It makes version from the past based on a copy approach: each version is a FAMIX model. It has some similarity with our idea, however, it is a copy-based approach, so it is impractical to perform interactive modifications. Another difference is that our idea is based on analysis of the future, Hismo is a study of the past. In [Gir05], the author proposes some metrics to compare which elements have changed. In our approach the goal is to have a notion of impact of a change.

Other models exist as [ALS09] which proposes a technique based on a probabilistic model, a Bayesian network is used to analyze the impact of an entry scenario. Orion is not concerned by this type of model because it provides the real modification of models.

Change assessment. [LO96] and [Lee98] propose an algorithm for analyzing of change impact with the detection of inheritance, encapsulation, and polymorphism. The algorithm proposes an order of changes based on the repercussion on self, children and clients. This method is the first one applied to an object model. It is also restricted to classes. Some approaches try to predict changeability, they assess the impact of a change to a code location by looking at previous change impact upon this location. This domain could be used in Orion to better reflect the impact of changes. [CH07] presents a decision-tree-based framework to assess design modularization for changeability. It formalizes design evolution problem as decision problems, model designs and potential changes using augmented constraint networks (ACN). [ACdL99] uses metrics comparison to try to predict the size of evolving Object-oriented systems based on the analysis of classes impacted by a change. A change is computed as a number of lines of code added or modified), but they do not provide the possibility to compare some versions and to choose one.

Other approaches propose change impact analysis based on test regression [RT01]. [KGH+94] proposes a change impact analysis tool for regression tests. In this paper they define a classification of changes based on inheritance, association and aggregation. They also define formal algorithms to compute impacted classes and ripple effects. The Chianti tool [RST+04] is able to identify tests which run over the changed source code. They can be run in priority to test regression in the system. For each affected test, Chianti reports the set of related changes. Orion cannot do this kind of analysis because it does not work on source code.

7.3. Change-based environment

Some models exist to support changes as a part of development. The Prism Model [Mad92] proposes a software development environment to support change impact analysis. This work introduces a model of change based on deltas that supports incremental changes. As it is based on deltas, it is not really possible to analyze different model in parallel.

Another work in change management system is Worlds ([WK08]). It is a language construct which reifies the notion of program state. The author notes that when a part of a program modifies an object, all elements which reference this object are affected. Worlds has been created to control the scope of side effects. With the same idea to control side-effect but restricted to source-code, ChangeBoxes [DGL+07] propose a mechanism to make changes as first-class entities. ChangeBoxes support concurrent views of software artifacts in the same running system. We can manipulate ChangeBoxes to control the scope of a change at runtime. Compared to orion which is structure-oriented, ChangeBoxes are used to integrate changes in a runtime environment.

Several other works [Joh88, MOR93] were previously done in this domain, but they manage only a single branch. Worlds manage several parallel universes. The limitation of Worlds is that it only captures the in-memory side effects. Compared to Orion, this work is a source code based approach, Orion is model-based. In the future, it could be possible to use this type of approach to expand Orion and populate changes on source code with mastering side effects.

The Model Driven Engineering domain and particularly Model transformation could have similar ideas with Orion approach. In [MG06], authors propose a taxonomy of Model Transformation for helping developers who want to choose a model transformation approach. In the enumeration of characteristics of a model transformation, there is no information about multiple models management. One more time, in the future, Orion could integrate a link to source code and have a bidirectional transformations mechanism, as Model transformation provides.

8. Conclusion and perspectives

We present a novel approach to reengineering through simulation of changes in source code model. In particular, we claim that software maintainers should be able to assess and compare multiple change scenarios. We define user and technical requirements for an infrastructure supporting our vision, ranging from reuse of existing tools to handling simultaneous versions in memory. We implemented the requirements in Orion, our current infrastructure.

The main concern detailed in this paper is the efficient manipulation of simultaneous versions in memory of large source code models. Copy approach does not scale up in memory for such models. In Orion, only changed entities are copied between versions, unchanged entities being effectively shared. Then, basic queries take care of retrieving a consistent view of entities in the analyzed version. Our benchmarks report the large gain in memory for our approach with an acceptable overhead in query running time. Overall, it allows Orion to scale up and be usable. We have started to use Orion on two large case studies.

In future work, we plan to optimize query running time by using optimized search structure for retrieving entities between versions. We envision dedicated visualizations for change assessment. We also need to assess how our current infrastructure handles new reengineering tasks: then new tools and models need to be plugged on top of Orion and possibly adapted. In particular, we believe Orion could be a useful approach to assess automatic reengineering tools such as [ADSA09]. Such tools usually provide a refactored model without rationale for their decisions, which makes reengineers wary of the result. With Orion, we could “watch” automatic algorithms iterate over the model, creating new versions at each important step, and giving better insights on the inner working of the tool. Overall, Orion aims to provide better decision support for software maintainers.

Acknowledgements. We gratefully acknowledge Nicolas Anquetil and Marcus Denker for their review.

References

[ACdL99] G. Antoniol, G. Canfora, and A. de Lucia. Estimating the size of changes for evolving object oriented systems: A case study. In *METRICS ’99: Proceedings of the 6th International Symposium on Software Metrics*, page 250, Washington, DC, USA, 1999. IEEE Computer Society.

[ADSA09] H. Abdeen, S. Ducasse, H. A. Sahraoui, and I. Alloui. Automatic package coupling and cycle minimization. In *International Working Conference on Reverse Engineering (WCRE)*, pages 103–112, Washington, DC, USA, 2009. IEEE Computer Society Press.

[ALS06] M. K. Abdi, H. Lounis, and H. A. Sahraoui. Analyzing change impact in object-oriented systems. In *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 310–319, Washington, DC, USA, 2006. IEEE Computer Society.

[ALS09] M. K. Abdi, H. Lounis, and H. A. Sahraoui. Analyse et prédiction de l'impact de changements dans un système à objets : Approche probabiliste. In *Proceedings of LMO'09*, 2009.

[BA96] S. A Bohner and R.S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[BMZ+05] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal on Software Maintenance and Evolution: Research and Practice*, pages 309–332, 2005.

[CH07] Y. Cai and S. Huynh. An evolution model for software modularity assessment. In *WoSQ '07: Proceedings of the 5th International Workshop on Software Quality*, page 3, Washington, DC, USA, 2007. IEEE Computer Society.

[CI90] E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[CKKL02] M. A. Chaumun, H. Kabaili, R. K. Keller, and F. Lustman. A change impact model for changeability assessment in object-oriented software systems. *Science of Computer Programming*, 45(2-3):155 – 174, 2002.

[DGKR09] S. Ducasse, T. Gîrba, A. Kuhn, and L. Renggli. Meta-environment and executable meta-language using Smalltalk: an experience report. *Journal of Software and Systems Modeling (SOSYM)*, 8(1):5–19, February 2009.

[DGL+07] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007.

[DMJN07] D. Dig, K. Manzoor, R. Johnson, and T. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *International Conference on Software Engineering (ICSE 2007)*, pages 427–436, 2007.

[DTD01] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[Ebr09] P. Ebraert. *A bottom-up approach to program variation*. PhD thesis, Vrije Universiteit Brussels, 2009.

[ESaSD08] Steve Easterbrook, Janice Singer, Margaret anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research, 2008.

[Gir05] T. Girba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Bern, Bern, November 2005.

[Han97] J. Han. Supporting impact analysis and change propagation in software engineering environments. In *STEP '97: Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97) (including CASE '97)*, page 172, Washington, DC, USA, 1997. IEEE Computer Society.

[Joh88] R. Johnson. TS: An optimizing compiler for Smalltalk. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, volume 23, pages 18–26, November 1988.

[KGH+94] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 202–211, 1994.

[LDDB09] J. Laval, S. Denier, S. Ducasse, and A. Bergel. Identifying cycle causes with enriched dependency structural matrix. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, Lille, France, 2009.

[LDDK09] J. Laval, S. Denier, S. Ducasse, and A. Kellens. Supporting incremental changes in large models. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST 2009)*, Brest, France, 2009.

[Lee98] M. Li Lee. *Change impact analysis of object-oriented software*. PhD thesis, George Mason University, Fairfax, VA, USA, 1998. Director-Jeff Offutt.

[LO96] M. Li Li and A. Jefferson Offutt. Algorithmic analysis of the impact of changes to object-oriented software. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 171–184, Washington, DC, USA, 1996. IEEE Computer Society.

[Mad92] N. H. Madhavji. Environment evolution: The prism model of changes. *IEEE Transaction in Software Engineering*, 18(5):380–392, 1992.

[MG06] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.

[MOR93] J. G. MORmSETT. Generalizing first-class stores. In *ACM SIGPLAN Workshop on State of Programing Language*, pages 73–87, New York, 1993. ACM.

[MT04] T. Mens and T. Tourwé. A survey of software refactoring. *Transactions on Software Engineering*, 30(2):126–138, 2004.

[NDG05] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[NMB05] T. Nguyen, E. Munson, and J. Boyland. An infrastructure for development of object-oriented, multi-level configuration management services. In *Internationl Conference on Software Engineering (ICSE 2005)*, pages 215–224. ACM Press, 2005.

[NMT05] T. Nguyen, E. V. Munson, and C. Thao. Managing the evolution of web-based applications with webscm. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 577–586, Washington, DC, USA, 2005. IEEE Computer Society.

[PPV00] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 345–355, New York, NY, USA, 2000. ACM.

[Rob08] R. Robbes. *Of change and software*. PhD thesis, University of Lugano (Switzerland), 2008.

[RST+04] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, oct 2004.

[RT01] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53. ACM Press, 2001.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[WK08] A. Warth and A. Kay. Worlds: Controlling the scope of side effects. Technical Report RN-2008-001, Viewpoints Research, 2008.