Supporting Incremental Change in Large System Models

Jannik Laval Simon Denier
Stéphane Ducasse
RMoD Team - INRIA - Lille Nord Europe - USTL
- CNRS UMR 8022, Lille, France
firstname.lastname@inria.fr

Andy Kellens
Software Languages Lab - Vrije Universiteit
Brussel, Pleinlaan 2, 1050 Brussels, Belgium
akellens@vub.ac.be

Abstract

When reengineering large systems, software developers would like to assess and compare the impact of multiple change scenarios without actually performing these changes. A change can be effected by applying a tool to the source code, or by a manual refactoring. In addition, tools run over a model are costly to redevelop. It raises an interesting challenge for tools implementors: how to support modification of large source code models to enable comparison of multiple versions. One naive approach is to copy the entire model after each modification. However, such an approach is too expensive in memory and execution time. In this paper we explore different implementations that source code metamodels support multiple versions of a system. We propose a solution based on dynamic binding of entities between multiple versions, providing good access performance while minimizing memory consumption.

1. Introduction

Software architecture evolution, change impact analysis, software quality prediction, remodularisation and so on, are important tasks in a reengineering process. They often require developers to be able to compare different versions of a system to pick the most adequate path of changes.

Let us consider a typical reengineering session: the reengineer might apply a visualization, then perform an analysis, apply other analyses or visualizations, then perform some refactorings. Afterwards, the reengineer would like to reapply the original visualization to assess whether the proposed changes had a positive impact on the system. While iterating several times, he may pick one version and roll-back to the previous one, reapply some changes and finally apply the changes to the actual software.

All these operations should be performed on the model of the source code. If we step back from this scenario we see that we must have a model of the software. It must allow one to compare different versions of a model and branch, modify, and navigate through these different versions.

The question is: how do we support these model manipulations (editing, comparison...) for large source code models and many small modifications? A naive idea is to make a copy of the original model and to modify it for each version. However, a lot of memory is wasted by copying non-modified elements. For example, modifying one package in a system with 100 packages requires make 99 useless copies.

In this paper, we present several possible high approaches to support simultaneous multiple versions of models. We discuss the pros and cons of each approach. As a compromise between memory consumption and the time needed to access elements within model versions, we have implemented an approach based on dynamic binding of entities across a tree of shared models.

This paper is organized as follows. Section 2 presents the running example used throughout the paper. Section 3 presents the chosen source code metamodel. Section 4 presents the design space we explored to keep track of multiple versions of a source code model, along with the pros and cons of each possible solution. Section 5 presents our solution. Section 6 discusses about our solution and future work. Finally, we conclude this paper in section Section 7.

2. Running example

Figure 1 presents an example system, which contains one package containing three classes (ClassA, ClassB and ClassC). Within the example we have four methods, such that method mA2() invokes mA1() and method mB2() invokes mB1().

Suppose we evolve the example system into a new version by applying the following changes: method mB1() is moved to ClassA; ClassC is removed. Note that these modifications also have an impact on the other entities in the model: ClassA needs to be aware that a new method was added; ClassB needs to know that method mB1() was removed and package Pack1 has to be notified that it no longer contains Class C.

We present in Section 4 a number of approaches that enable keeping track of versions of a source code model. Based on the above example we discuss the pros and cons of each of these approaches.

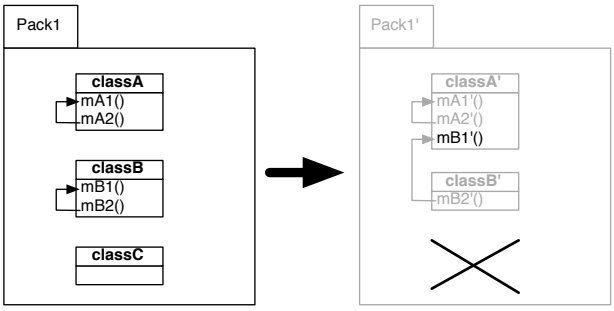

Figure 1. On the left is the original model; on the right a new version cred by moving mB1() and deleting classC.

3. FAMIX

FAMIX is a language independent metamodel. It describes the static structure of software systems, particularly object-oriented software systems. A class-diagram representation is available at `http://moose.unibe.ch/docs/famix/famix3.0`.

FAMIX is based on the meta-metamodel FM3 (Fame Meta-metamodel) which is a superset of a subset of EMOF [KV08]. Since a discussion of the FAMIX metamodel lies outside the scope of this paper, we refer the interested reader to [DTD01].

FAMIX has a lot of analysis tools thanks to the Moose Framework and Mondrian visualization tools. It allows us to make visualization, to compute metrics and provides some more complex tools as Hismo ([GÔ5]) or Dynamix ([Gre07]).

Since we place our work in the context of the FAMIX metamodel, we apply the following constraints:

- We need a change model which does not impact other aspects of the FAMIX metamodel such as metrics, relationships, and so on. Each version of a model should be usable with any tool based on FAMIX, in order to maximise the reuse of existing tools.
- It should also allow reengineers to compare versions.

Figure 2 represents a FAMIX model of Figure 1 example. All classes and methods have a parent package. Associations between entities (invocations, variable accesses, inheritance links and class references) are reified and are explicitly known by the two entities involved.

An important feature of FAMIX 3.0 is that all relationships are bidirectional. So, attribute automatically updates the other side of the relationship (opposite) entities. For example, when mB1() is moved from ClassB to ClassA, this information is propagated in mB1(), ClassA and ClassB.

In Figure 2, the dotted line represents the link between mB1() and ClassB in the left of Figure 1. The bold lines represent modifications done in the second version: mB1() is now in ClassA and ClassC is removed.

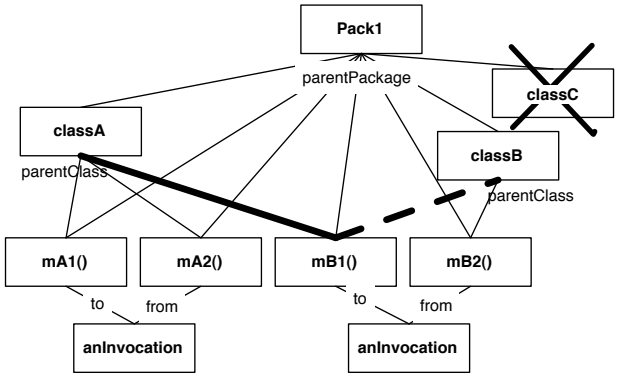

Figure 2. Famix representation of Figure 1.

4. Approaches for modeling multiple modifications of a FAMIX model

There are multiple ways of representing different versions of a particular FAMIX model. Four naive approach are presented in this section:

- Copy Approach, where the entire model is copied to a new model.
- Delta Approach, which keeps track of the delta (changes) between the original and the new model.
- a LookUp Approach based on creating a partial copy of the model.
- a mixed LookUp/Copy Approach also based on creating a partial copy of the model.

We now present each approach and analyze its performance. This analysis is based on three important criteria: (1) creation time of a version, (2) access time to a versioned entity, and (3) memory usage.

All these approaches have to work with Famix tools. So, an analysis of the integration of each approach is made.

For our analysis, we consider the following parameters:

- m, the number of entities in the original model;
- v, the total number of versions between the original model and the version of model that is being accessed;
- d, the number of modifications in a version including the impacted elements.

For all of these possible approaches, we consider three types of modifications: removing an entity or an association, adding an entity or an association, and moving an entity between containers (class, package...).

4.1 Copy Approach

The Copy Approach approach is the most intuitive and straightforward. For each version of the model, a copy of the previous model is created and the modifications are applied to this copy (Figure 3). In other words, each version is a complete model and is totally autonomous.

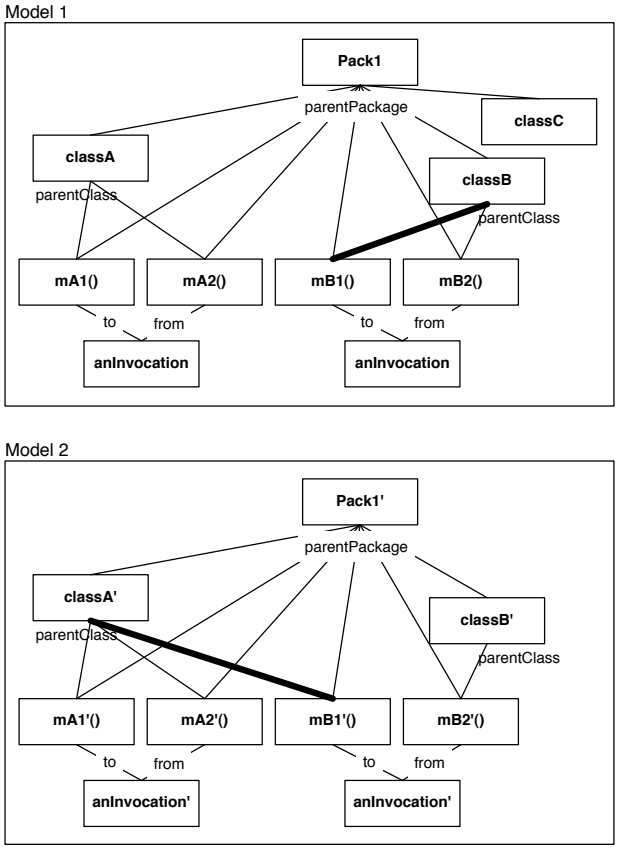

Figure 3. The example represented in the Copy Approach

With Copy Approach, each version is represented by a model that is independent of previous versions and from the history of changes. This implies that it can be used regardless of the change history and it can be used by all tools that are based on FAMIX. This independence of the history hides the modifications that were applied to the model. Since a version is a complete model, the modifications that were made are no longer visible, so to to know them, it is necessary to add a marker as a set of change for each version as in Section 4.2.

The most important problem of this approach is the duplication of entities that are not modified. In Figure 3, all entities are copied whereas only three entities are modified by the changed link (ClassA, ClassB and mB1()). This has a negative impact on both the time needed to create the model and the memory consumption of various versions of the model.

Performance analysis

- Creation time: $O(m + d)$. Each version is created with n entities of the previous version and d modifications.
- Access time: $O(1)$. To access an element, this approach is fast. It can immediately access any entity in any version of the model.
- Memory usage: $O(m * v)$. Each version is complete and the n elements are duplicated in all m versions. This represents a high overhead if only a small number of changes were made.
- Famix integration: Copy Approach will work with all Famix tools, because each model is a Famix Model.

4.2 Delta Approach

The Delta Approach uses a list of changes that are applied to a Famix model (Figure 4). Consequently, a version is either a Famix model, or a list of changes to the previous version of the model.

In Figure 4 we see the Delta Approach applied to our running example. Model1 is the original model, Model2 is represented by a list of changes. There are three modifications which remove the link between mB1() and ClassB, create the link between mB1() and ClassA, and remove ClassC.

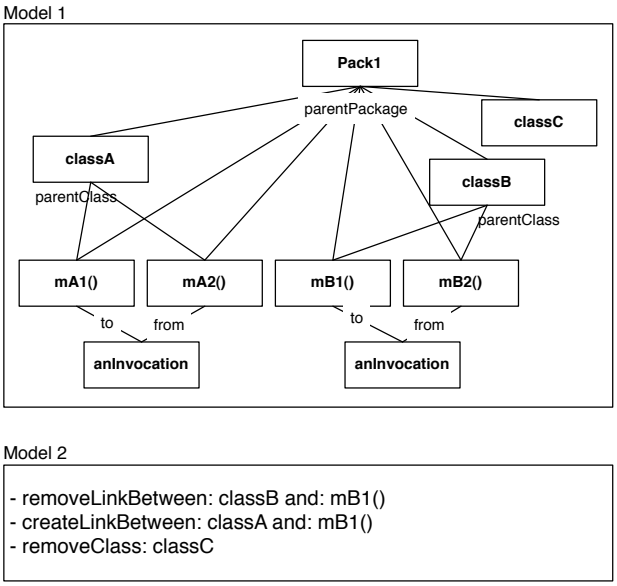

Figure 4. The example represented in Delta Approach

The major advantage of this approach is that it keeps track of the history of changes. Only one model is stored in the system, and changes are applied on it. For the first couple of versions, this method is fast because the set of modifications remains relatively small. But the larger the number of version is, the slower the access of the entities in a later version will be since all previous changes starting from the original model need to be applied.

For example, when comparing two versions this can become a problem. As there is only one full model of the system (the others are represented as lists of changes), the comparison of two versions is an expensive operation: all the changes must be replayed. A possible solution to bypass this is to make and cache a copy of the model on demand for the situations that we want to compare. This solution is interesting when there are more modifications than versions we want to compare, otherwise the same problems occur as in the Copy Approach.

Performance analysis

- Creation time: $O(d)$. A version is a list of changes, thus creating a model simply means keeping track of these changes.
- Access time: $O(n + m * d)$. To access to a version, it is necessary to build all versions in between the original version up to the requested version using the changes. This is an expensive operation.
- Memory usage: $O(n+m*d)$. Each version contains only a list of changes. This result is small in memory consumption since only the necessary information is stored.
- Famix integration: Each time a version should be accessed, a Famix Model is created based on the original model and changes. So Famix tools works by default on a version.

4.3 LookUp Approach

The objective of the LookUp approach is to avoid copying the entire model for each version and thus providing support for incremental models.

The LookUp approach copies entities only if they were changed between versions. Each version keeps a pointer to the previous version. This pointer is necessary to ensure that information is retrieved with respect to the current version that is being queried.

In Figure 5, Model2 is defined as all elements that were altered in Model1. All other elements are retrieved via the lookup mechanism. Each time a request for information is being handled, the lookup of the entity is performed starting at the current version. If that entity is not present in the current version, it will be requested from the parent version and so on. For example, if we request mB2'().parentClass, the result is ClassB': method mB2() is not part of Model2 and will be retrieved via the pointer to the previous version from Model1. However when asking method mB2() for its parent class, this lookup will start again in Model2, thus returning the correct class. The semantics is similar to the one of self in OOP since the lookup always starts from the current model.

Two problems however appear in this approach: the first one is the cost of performing a lookup. The larger the number of versions is, the slower the lookup possibly becomes because in the worst case the lookup must traverse the entire set of models. The second problem is the case of deletion of an entity: if an entity is removed from a version, the lookup will check in the previous version. To remedy this problem, we must create an entity which represent such a special case and stop or adapt the lookup when such an entity is encountered. For example, Figure 5 shows that for our running example, we have to provide a special entity removedClassC in Model2 in order to ensure that performing a lookup for class C does not return the version of the class from the previous model.

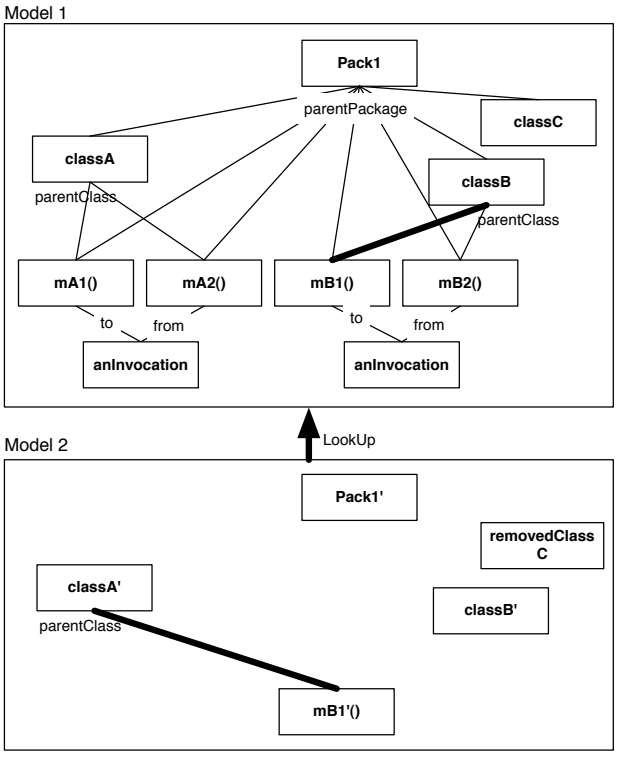

Figure 5. LookUp Method

Performance analysis

- Creation time: $O(d)$. As with all the partial copy approaches, only modified elements are stored in a version.
- Access time: $O(v)$. An entity is accessed by searching from the most recent version up to the first version of the system.
- Memory usage: $O(m + d * v)$. As with the previous two approaches, the memory usage of LookUp is small as only modified entities and these containers are stored.
- Famix integration: This approach is more complex than the previous. When some requests are made, this approach switch between several versions. So a modification of accessing methods of Famix is necessary to have correct information related to a specific version.

4.4 Hybrid approaches

Each of previous approaches exhibits a number of flaws with regards to the requirements we set forth. Copy Approach requires a copy of all (non-modified) elements, thus resulting in excessive memory usage. While Delta Approach minimizes the memory usage, it requires a version to be generated by applying an entire change history, which is an expensive operation. The Lookup approach resolves this flaw, but has a poor time complexity for looking up an entity.

In this section, we propose a number of hybrid approaches to bypass these problems. Two such approaches seem valuable and are introduced here, namely Lookup/Copy and Lookup/Pointer.

Lookup/copy approach. In this approach, each changed element along with all the elements it is related to are completely copied when creating a new version of the model. This approach allows one to make partial copies of impacted elements and to perform a lookup with non-modified elements. In our example (Figure 2), ClassA, ClassB and mB1() are copied, along with all linked elements. This represents a complete copy of the model.

As a downside, this model introduces a high probability that the whole model will get copied. Since FAMIX uses bidirectional links between elements, modifying one element modifies also all linked elements, which can result in a cascade of necessary copy operations. If there are groups of entities without connection, which happens rarely in a software, the model is not completely copied. So when a copy of an entity is done, this approach is slower than the copy approach because the process collect entities in relation with the copied one and make the same process for these entities, etc.

An hybrid Pointer/Copy approach suffers from the same problem: all touched elements need to be copied. The difference still lies in how accesses to non-modified entities are handled. With a pointer approach the non-modified entity access is faster than with lookup approach.

Lookup/Pointer approach. The second hybrid approach is Lookup/Pointer. As in Section 4.3, this approach is based on copying only the changed entities. The unchanged elements are represented by pointers on the previous version. The lookup is used to get the version of an entity in the current model.

For example, in Figure 6:

- In Model1, ClassB.allMethods returns #(mB1() mB2()).
- In Model2, ClassB'.allMethods returns #(mB2()).
- In Model1, mB1().parentClass returns ClassB.
- In Model2, mB1'().parentClass returns ClassA'.
- In Model2, mB2'().parentClass returns ClassB', which is the version of ClassB in Model2.

In fact, in the case of indirect request, the approach tests if the requested entity is in the current model. An indirect request is a request made through another element of the model. This is a request of entities in relation with the entry entity. For example "FamixMethod » parentClass" goes through a FamixMethod to give its parent class. In Figure 6, mB2().parentClass represent mB2() as the entry entity and return the result of parentClass as an indirect request. Indirect request is problematic because the returned result can be entities in a previous model, which must be updated to the current version. In the case of direct request, there is no problem because in a model, if the entity exists, it is in the correct version, and if it does not exist (no entity, no pointer), this is a removed entity.

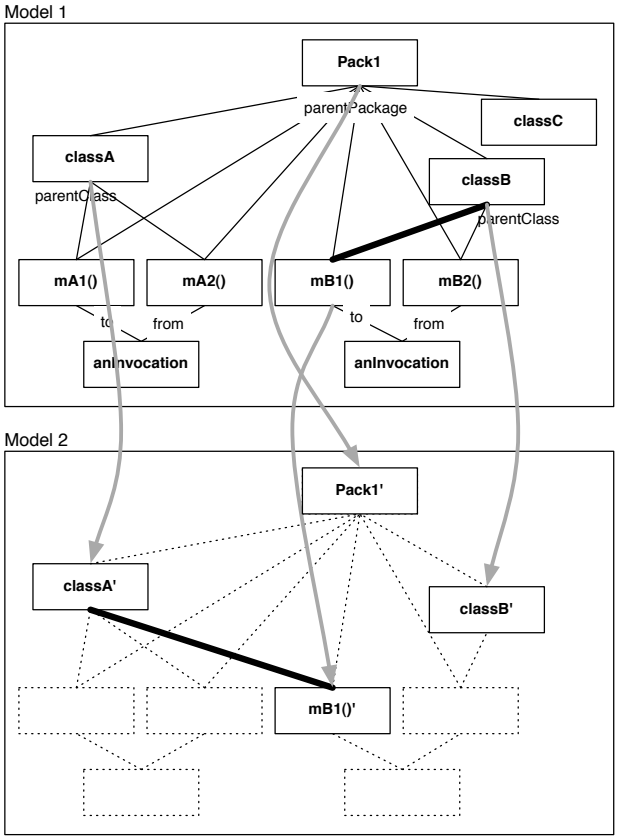

Figure 6. LookUp/Pointer Approach

Performance analysis

- Creation time: $O(d)$. The creation time is the same as lookUp approach.
- Access time: $O(v)$. While the access time is not as good as a Copy of the entire model (which was $O(1)$), it is considerably better than the worst case scenario that is offered by the Lookup approach ($O(m)$).
- Memory usage: $O(m + d * v)$. This is the minimum size necessary for keeping track of all versions.

- Famix integration: The integration with Famix tools has the same problems than lookUp approach. A modification of accessing method is necessary.

5. Orion

Based on our assessment, the Lookup/Pointer approach seems to best satisfy the requirements we put forward before. As a proof-of-concept, we have implemented such a scheme for keeping track of multiple versions of a model by means of an extension to the FAMIX metamodel. This section explains this proof-of-concept named Orion. We explain how it works and what the benefits are of the approach.

5.1 Presentation

The Orion metamodel is an extension of FAMIX metamodel. All elements used in Orion are inherited from FAMIX. To make it easy to distinguish between Orion elements and FAMIX elements, we use the following naming convention: FAMIX entities are prefixed 'FAMIX' while Orion entities are prefixed 'Orion'. *e.g.,*, the Orion version of the FamixClass class is called OrionClass. For now, only a subset of elements are implemented: OrionPackage, OrionNamespace, OrionClass, and OrionMethod.

Orion differs from the FAMIX metamodel in two ways. First, in order to ensure that when creating a new version of a model the old version remains consistent, Orion entities do not automatically update their link to other entities as is the case with the FAMIX3.0 model (explained in Section 3). In our example, the replacement of ClassA by ClassA' in Model2 should not modify Pack1, mA1() and mA2(). In FAMIX3.0, these elements would have been automatically updated, resulting in an inconsistent model. The second difference is that in contrast to FAMIX3.0, accessing entities in a model is not done directly, but via a binding process. This process is explained in Section 5.3.

5.2 Orion metamodel

Since Orion is an extension of Famix, OrionElement inherits from FamixEntity and OrionModel inherits from MooseModel. The metamodel in Figure 7 is simplified, in the complete one OrionPackage inherits from FamixPackage, OrionClass from FamixClass, etc.

OrionElement has two subclasses: OrionEntity and OrionAssociation. OrionEntity represents structural entities in the model. Currently, we support four such entities: OrionClass, OrionMethod, OrionPackage, OrionNamespace. OrionAssociations represents reified associations between OrionEntities. Yet again, we support for such associations: OrionReference, OrionInvocation, OrionInheritance, OrionAccess. All OrionElements have a OrionID which is a unique identifier for an entity. It is used in the binding process (Section 5.3).

OrionModel inherits from MooseModel, it represents a version of model, so it contains Orion elements and has a

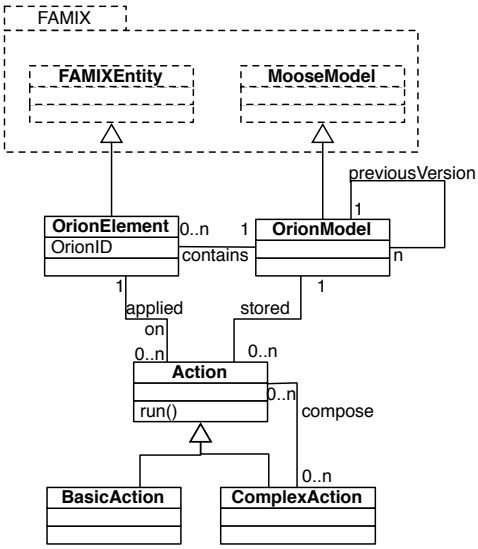

Figure 7. the Orion Metamodel, extension of Famix

pointer to the previous version from which it was derived. Note that the first OrionModel does not have such a parent pointer (as it is the model containing all entities belonging to the initial model).

The class Action and its subclasses represents changes that can be applied to a model. It is possible to compose some actions to make a complex one. Actions are stored in the OrionModel to keep an history of changes, as in a Delta Approach (Section 4.2). Note that this allows us to not only keep track of the various versions of a model, but also gives us access to a full history of the changes between models, expressed in terms of Actions.

5.3 Binding process

Each OrionElement contains an OrionID. It represents the unique identifier for an entity in a model, but is the same for all versions of an entity. In Orion, we use this identification to trace versions of an element in different model versions. If a new entity is added to a model, a new OrionID will be created for this entity. However, in the case an existing entity is altered between versions of a model, then the altered version of the entity will share the same OrionID as the original entity that was altered. This usage of the OrionID lies at the core of the binding process.

The binding process itself is fairly simple. When looking for a particular entity in a version, there are two possibilities. Either the entity is present in the model. This is the case when a new entity was added to a model, or an entity was altered between two versions of the model and the altered version is added in the (derived) model. Looking up such an entity is simple: the entity can directly be returned from the model. If the entity is not present in the model that is being searched, a binding between the entity and its version in

the correct version is initiated. In this version, the entity is looked up using the OrionID that serves as the unique identifier for the entity that is being retrieved. Similarly, when querying a particular entity, the same binding mechanism is used to ensure that the right version of an entity is returned. For example, in Figure 6 requesting mB2'().parentClass(), a binding must be made to have the latest version of the resulted entity. The aim is to obtain ClassB', not ClassB. The approach takes the OrionID of the entity and check it in the current version.

Using the same identification for all versions of an entity allows us to have history of an entity when this identification is coupled with the suite of versions.

5.4 Performance analysis of Orion

- Creation time: $O(d)$. When creating a new model, only the delta with respect to the previous version is calculated. As in lookup approach, the creation time is minimal.
- Access time: $O(1 + \delta)$. If an entity is in the current version it returns the result in O(1). If not, it retrieves the updated entity with the corresponding OrionID. This may take a little more time because the process concerns two versions of an entity: the old and the updated.
- Memory usage: $O(m + d * v)$. If we would like to have possible accesses to all version, it is necessary to have the original model and all modifications.
- Famix integration: As all approaches which partially copy models, this approach must modify some accessing methods of Famix. As this approach is an extension of Famix, these methods has been written in OrionElements and take account the binding process. So Famix tools works with a version of this approach like with a Famix Model.

The theoretical evaluation above shows that in comparison to the other (Table 1) approaches, Orion provides an interesting trade-off between model creation time, access time and memory consumption. While the binding of an entity is not performed in constant time, our implementation does outperform all other approaches particularly for the cost of access which is slower than the copy approach. Furthermore, our approach minimizes the amount of memory that is necessary to keep track of the evolution of a model.

Approach	cost of creation	cost of access	memory cost
Copy	O(m+d)	O(1)	O(m*v)
Delta	O(d)	O(m + v*d)	O(m + v*d)
Lookup	O(d)	O(v)	O(m + v*d)
Lookup/pointer	O(d)	O(v)	O(m + v*d)
Orion	O(d)	O(1 + δ)	O(m + v*d)

Table 1. Performance of approaches

6. Discussion

Orion approach has been implemented above Famix Metamodel on Pharo Smalltalk. All tools of Moose platform work with Orion. It includes metrics, visualization and browser. For each visualization, we could make some new feature interact with the visualization to change the model. As in eDSM [LBD09], we could see the impact when a cycle between packages is broken.

The analysis of cost made in this paper is theoretic. The first future work will be a case study to validate these values. We must compare the different approaches on a complexe software model. A second case study should be made to analyze the usefulness of comparing multiple versions. We think that when a software reengineer works, he should have several possibilities of future structure and makes choice based on the comparison of versions. We want to know in average how many versions a software engineer in practice needs. We expect to get less than 10 versions.

The use of algorithms for optimizing the software structure can be implemented with Orion. A lot of algorithms makes structure changes, but they provide only the final result model. Coupled with Orion, a scenario could produce several versions, stressing the problems presented in this paper.

The integration with Famix/Moose tools is important. All tools developed on Moose works with Orion. We could make some new feature to interact with the visualization and make different changes of the model.

7. Conclusion

In this paper, we have compared several ways of supporting simultaneous versions of a model. We showed that naive approaches as copy or delta are expensive in creation time, in memory or in time to access an entity. The Lookup Approach appears not adapted to Famix. But each of these approaches has some benefits. Based on this analysis, we developped mixed approach based on dynamic bindings.

The Orion approach, which uses pointers and bindings, offers a good compromise between memory consumption and access time at execution. This approach, supporting Famix metamodel, can build multiple versions of a software model with a limited computing cost.

This approach can be used to support tools that analyze several versions of possible future system. Coupled with manual or automatic change recommandation system, it can offer a new tool for choosing how to modify a system. Future improvements will go in this direction with possibilities to analyze impacts of changes proposed by Moose tools and optimization algorithms.

Acknowledgement. we gratefully thank Andrew P. Black for his deep review.

References

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

[GÔ5] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Bern, Bern, November 2005.

[Gre07] Orla Greevy. Dynamix — a meta-model to support feature-centric analysis. In *Proceedings of FAMOOSr 2007 (Ist International Workshop on FAMIX and Moose in Reengineering)*, June 2007.

[KV08] Adrian Kuhn and Toon Verwaest. FAME, a polyglot library for metamodeling at runtime. In *Workshop on Models at Runtime*, pages 57–66, 2008.

[LBD09] Jannik Laval, Alexandre Bergel, and Stéphane Ducasse. Matrice de dépendances enrichie. In Bernard Carré and Olivier Zendra, editors, *Actes des journées Langages et Modèles à Objets*, volume RNTI-L-3 of *Revue des Nouvelles Technologies de l'Information*, pages 107–122, Nancy, March 2009. Cépaduès-Editions.