JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2006; **18**:207–236 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Research

Modeling History to Analyze Software Evolution

Tudor Gîrba[1†*] and Stéphane Ducasse[2‡]

[1] *Software Composition Group, Institute for Applied Mathematics and Computer Science, University of Berne, Neubrückstrasse 10, CH-3012 Berne, Switzerland*
[2] *Language and Software Evolution Group, Listic, Université de Savoie, BP 806-F74016 Annecy Cedex, France*

SUMMARY

The histories of software systems hold useful information when reasoning about the systems at hand or when reasoning about general laws of software evolution. Over the past 30 years more and more research has been spent on understanding software evolution. However, the approaches developed so far do not rely on an explicit meta-model, and thus, they make it difficult to reuse or compare their results. We argue that there is a need for an explicit meta-model for software evolution analysis. We present a survey of the evolution analyses and deduce a set of requirements that an evolution meta-model should have. We define, Hismo, a meta-model in which *history* is modeled as an explicit entity. Hismo adds a time layer on top of structural information, and provides a common infrastructure for expressing and combining evolution analyses and structural analyses. We validate the usefulness of our a meta-model by presenting how different analyses are expressed on it.

KEY WORDS: Software evolution, meta-modeling, history, reverse engineering, evolution analysis.

1. INTRODUCTION

During the 1970's it became clear that keeping track of software evolution was important, at least for very pragmatic purposes such as undoing last changes. Early versioning systems like the Source Code Control System (SCCS) made it possible to record the successive versions of software products [1]. This led to the usage of text-based delta algorithms for understanding where, when and what changes appeared in the system [2]. Some basic services were also added

*Correspondence to: Neubrückstrasse 10, CH-3012 Berne, Switzerland–girba@iam.unibe.ch
Contract/grant sponsor: Swiss National Science Foundation; contract/grant number: "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).
†E-mail: girba@iam.unibe.ch
‡E-mail: stephane.ducasse@univ-savoie.fr

Received February 18, 2005
Revised December 20, 2005
Accepted December 20, 2005

to model extra or meta information such as who changed files and why. However only very rudimentary models were used to represent this information – typically a few unstructured lines of text to be inserted in a log file.

While versioning systems enabled recording the history of each source file independently, configuration management systems attempted to record the history of software products as a collection of versioned source files. Research on configuration management was very active in the 80's and 90's, but the emphasis was still on controlling and recording software evolution.

The importance of *modeling and analyzing* software evolution started to be recognized in the early 1970's with the work of Lehman [3]. Yet, it was only until recent years that extensive research has been spent on exploiting the wealth of information residing in versioning repositories for different purposes like reverse engineering or cost prediction. Problems like software aging [4] and code decaying [5] gained increasing recognition both in the academia and in the industry.

Various approaches have been proposed to analyze aspects of software evolution for purposes like identifying driving forces in software evolution, or like reverse engineering. Each of these approaches typically focuses on only some traits of software evolution, and most of these approaches rely on ad-hoc models (i.e., models that are not described by an explicit meta-model), or their meta-model is specific to the goals of the supported analysis.

By a *meta-model* we understand a specification model for a class of systems under study where each system under study in the class is itself a valid model expressed in a certain modeling language [6]. By *model* we understand a simplification of a system built with an intended goal in mind [7].

A meta-model describes the way the domain can be represented by the model, that is, it provides bricks for the analysis. An explicit meta-model allows for understanding those bricks. Understanding the bricks allows for the comparison of the analyses built on top. Without an explicit meta-model, the comparison and combination of results and techniques becomes difficult [8].

The main drawbacks of the current approaches reside in the implicitness of their meta-model. There is no *explicit entity* to which to assign the evolution properties, and because of that it is difficult to combine the evolution information with the version information. Often no semantical units are represented (e.g., packages, classes or methods), therefore, there is no information about what exactly changed in a system. For example, it is difficult to identify large classes which did not change recently, while, as we will show, it is expressible in a simple fashion with our approach.

In this article we describe our approach to address the problem of providing a meta-model for software evolution analysis. We define Hismo, a meta-model centered around the notion of history as a *first class entity* (i.e., an explicit entity). We show how we build it both for structural entities (e.g., files, classes) and for structural relationships (e.g., inheritance). We use the relationships between structural entities (e.g., a class *has* methods) to build relationships between the corresponding history entities (e.g., in the history of a class *there were* methods with different histories). Furthermore, we also model relationships between histories based on change conditions (e.g., co-change relationships between modules).

In Hismo, time information is added on top of the structural information: The structural information can exist without any reference to history but can still be manipulated in the

context of software evolution. In other words, Hismo can be built on top of any snapshot meta-model without interfering with the existing meta-model. With Hismo we can reuse the analyses at structural level and extend them in the context of evolution analysis.

To show the expressiveness of Hismo we describe how several software evolution analyses can be expressed on it. As a simple use of Hismo, we define different measurements for histories which describe how software artifacts evolved. We present different examples of historical measurements and history manipulations and show different reverse engineering analyses we have built over the years [8, 9]: Yesterday's Weather [10], History-based detection strategies [11], and Class Hierarchy Evolution Visualization [12]. Furthermore, we also show how Hismo can be used to express analyses like historical co-change [13]. Each of these examples exercise different parts of Hismo.

Hismo is implemented in Van, a tool built on top of the Moose reengineering environment [14]. Moose supports the integration of different reengineering analyses by making their meta-model explicit [15]. Van implements several evolution analyses (like those in Section 4). We briefly sketch the implementation of Van focusing on how the usage of Hismo allows for the different analyses to be combined from the implementation point of view. For example, we show how we use two other tools (CodeCrawler and ConAn) for building evolution analysis.

The contributions of this article are: (1) the analysis of the requirements of the different types of information and their manipulation for evolution analyses, (2) the description of Hismo a meta-model centered around notion of history as a first class entity, and (3) the validation of this meta-model to support evolution analyses.

Article structure. In Section 2 we analyze the current state of the art on software evolution approaches and determine a list of requirements that a meta-model should meet. We introduce Hismo, our meta-model in Section 3. As a validation for our approach we define measurements for quantifying changes, and we present some reverse engineering analyses enabled by our meta-model (Section 4). In Section 5 we discuss how our meta-model compares with the requirements we gathered. Section 6 deals with the details of our implemented tools. A glossary of terms we use throughout the article can be found in Section 7.

2. ANALYZING SOFTWARE EVOLUTION APPROACHES AND THEIR UNDERLYING META-MODELS

In this section we review different approaches for analyzing software evolution, the goal being to identify the requirements of the different analyses from the point of view of a common evolution meta-model. The most straight forward way to gather the requirements would be to analyze the different meta-models. Unfortunately, in most of the cases, the meta-models are not detailed (most of the time they are not even mentioned). In these cases we infer the minimal meta-models required for the particular analysis.

From our literature survey we identified two major categories of approaches depending on the granularity level of information representation: *version-centered* approaches and *history-centered* approaches.

Version-centered approaches consider version as a representation granularity, while the history-centered approaches consider history as representation granularity. For example, a

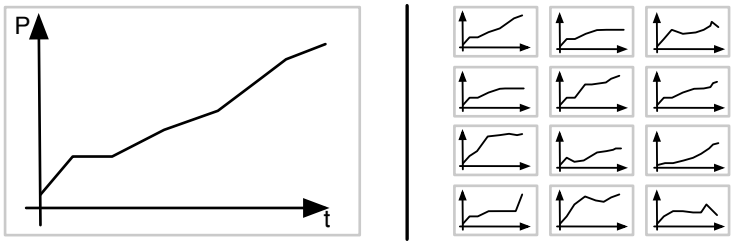

Figure 1. The evolution chart shows a property P on the vertical and time on the horizontal. It is suitable to use it to look at one property evolution in one entity (left side), but it is difficult to use it for comparisons of multiple property evolutions (right side).

graphic plotting the values of a property in time is a version-centered approach; on the other hand, a measure of how old is a file is a history-centered approach.

Following, we discuss these approaches and we summarize at the end of the section with a list of requirements for an evolution meta-model.

2.1. Version-centered approaches

The *version-centered* analyses use a version as a representation granularity. They have as target answering the question of when something happened in the history. In the next paragraphs we take a look at several such approaches and focus on three representative approaches: Two Version Comparison, the Evolution Chart, and the Evolution Matrix.

2.1.1. Two versions comparison

Comparing two versions is the base of any evolution analysis. We enumerate here three approaches that focus on finding different types of changes.

Demeyer *et al.* used the structural measurements to detect refactorings like rename method, or move method [16]. They represented each version with a set of metrics, and then identify changes based on analyzing the change in the measurements.

Xing and Stroulia detected several types of changes between two versions [17]. They represented each version of the system in an XMI format and then applied UML Diff to detect fine-grained changes like: addition/removal of classes, methods and fields; moving of classes, methods, fields; renaming of classes, methods, fields. Several applications have been based on this approach [17, 18, 19]. We discuss them in Section 2.2.3.

Antoniol and Di Penta used the similarity in vocabulary of terms used in the code to detect refactorings like: rename class, split class, or merge class [20]. They represented versions of classes with vectors holding the relevance of the different terms used in the system for the

particular class, and they compare the distance between the vectors of different versions to detect the refactorings.

Requirement: *An evolution meta-model should provide detailed information at different levels of abstraction for understanding the changes.*

2.1.2. Evolution chart

Since 1970 research is spent on building a theory of evolution by formulating laws based on empirical observations [21, 22, 3, 23, 24]. The observations are based on the interpretation of evolution charts which represent some property on the vertical axis (e.g., number of modules) and time on the horizontal axis (see Figure 1). Gall *et al.* employed the same kind of approach while analyzing the continuous evolution of the software systems [25]. Recently, the same approach has been used to characterize the evolution of open-source projects [26, 27, 28].

This approach is useful when we need to reason about the evolution of a single property, but it makes it difficult to reason in terms of more properties at the same time, and provides only limited ways to compare different property evolutions (by property evolution we denote how a particular property evolved in an entity). That is why, typically, the charts are used to reason about the entire system, though the chart can represent any type of entity.

Requirement: *An evolution meta-model should provide for comparison of property evolutions.*

In Figure 1 we give an example of how to use the evolution charts to compare multiple entities. In the left part of the figure we display a graph with the evolution of a property P of an entity – for example it could represent number of methods in a class (NOM). From the figure we can draw the conclusion that P is growing in time. In the right part of the figure we display the evolution of the same property P in 12 entities. Almost all graphs show a growth of the P property but they do not have the same shape. Using the graphs alone it is difficult to say which are the differences and if they are important. Furthermore, if we want to correlate the evolution of property P with another property Q, then we have an even more difficult problem, and the evolution chart does not ease the task significantly.

Requirement: *An evolution meta-model should provide for combination of property evolutions.*

2.1.3. Evolution Matrix visualization

Visualization has been also used to reason about the evolution of multiple properties and to compare the evolution of different entities. Lanza and Ducasse arranged the classes of the history in an Evolution Matrix shown in Figure 2 [29]. Each rectangle represents a version of a class and each line holds all the versions of that class (the alignment is realized based on the name of the class). Furthermore, the size of the rectangle is given by different measurements applied on the class version. From the visualization different evolution patterns can be detected such as continuous growth, growing and shrinking phases etc.

With this visualization, we can reason in terms of two properties at the same time, and we can compare different evolutions. The drawback of the approach resides in the implicitness of the meta-model: there is no explicit entity to which to assign the evolution properties. Because

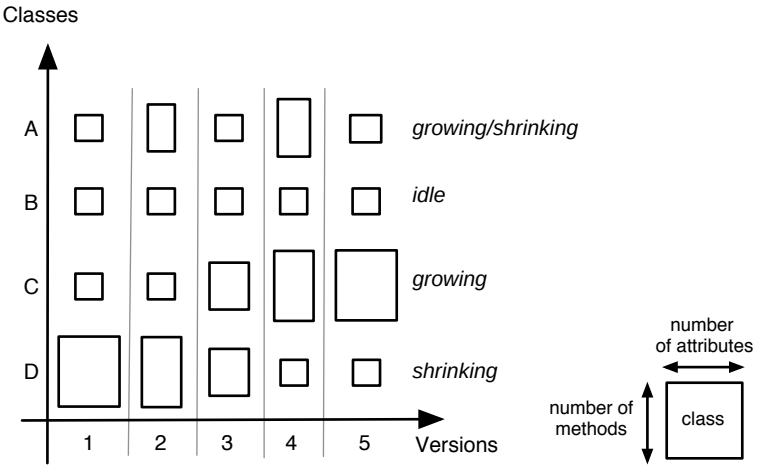

Figure 2. The Evolution Matrix shows versions nodes in a matrix. The horizontal position is given by the version number, and the vertical position is given by the name of the entity. The size of the nodes is given by structural measurements.

context Class

/ should return true if the class is large and if it was detected as being growing */*
derive isGrowingLargeClass: self.isLargeClass() & self.wasGrowing()

Figure 3. Example of a historical query written in OCL.

of that it is difficult to combine the evolution information with the version information. For example, we would like to know if the growing classes are large classes.

The code in Figure 3 written in the Object Constraint Language (OCL), shows how we would like to be able to put in one single automatic query, both evolution information (self.wasGrowing()), and structural information (self.isLargeClass()). We would only be able to express this if self would know both about the structure and about the evolution.

Requirement: *An evolution meta-model should provide for combination of property evolutions and snapshot properties.*

Another drawback here is that the more versions we have, the more nodes we have, the more difficult it gets to detect patterns when they are spread over a large space.

2.1.4. Other version-centered approaches

Other analyses are based on similar implicit meta-models and they require the same features from a meta-model.

Burd and Munro defined a set of measurements to quantify the dominance relations which are used to depict the complexity of the calls. They correlated the changes in these measurements with the types of maintainability activities [30].

Taylor and Munro visualized file changes with a technique called *revision towers* [31]. The purpose of the visualization was to provide a one-to-one comparison between changes of two files over multiple versions.

Rysselberghe and Demeyer used a scatter plot visualization of the changes to provide an overview of the evolution of systems and to detect patterns of change [32]. Jingwei Wu *et al.* used the spectograph metaphor to visualize how changes occur in software systems [33]. They used colors to denote the age of changes on different parts of the systems.

Jazayeri analyzed the stability of the architecture by using colors to depict the changes [34]. From the visualization he concluded that old parts tend to stabilize over time. Eick *et al.* proposed multiple visualizations to show changes using colors and third dimension [35].

Chuah and Eick proposed a three dimensional visualizations for comparing and correlating different evolution information like the number of lines added, the errors recorded between versions, number of people working *etc.* [36]. Holt and Pak proposed a detailed visualization of the old and new dependencies between modules [37]. Gulla proposed visualizations of C programs to detect changes in structure and in dependencies [38].

2.1.5. Discussion of version-centered approaches

The version-centered models allow for the comparison between two versions, and they provide insights to when a particular event happened in the evolution (e.g., a class grew instantly). The visual technique is to represent time on an axis and place different versions along this axis and make visible where the change occurred (e.g., using color, size, position).

Some of the analyses also used version-based techniques to compare the way different entities evolved over time. For example, the evolution chart was used to compare the evolution of different systems to detect patterns of change like continuously growing systems. The Evolution Matrix was also used to detect change patterns like growing classes or idle classes (i.e., classes that do not change). A major technical problem is that the more versions we have the more information we have to interpret.

Furthermore, when patterns are detected, they are attached to structural entities. For example, the authors said that they detected growing and idle classes. If we take a closer look at the Evolution Matrix, we see that it is conceptually incorrect because a class is just one rectangle while growing and idle characterize the entire line and not just one rectangle. That is, we can say a class is big or small, but growing and idle characterize the way a class evolved. From a modeling point of view, we would like to have a reification to which to assign the growing or idle property: the history as an encapsulation of evolution.

2.2. History-centered approaches

History-centered approaches have history as an ordered set of versions as representation granularity. In general, they are not interested in when something happened, but they rather seek to answer what happened and where it happened. In this approaches, the individual versions are no longer represented, they are flattened.

The main idea behind having a history as the unit of representation is to summarize the evolution according to a particular point of view. History-centered approaches often gather measurements of the history to support the understanding of the evolution. However, they are often driven by the information contained in the repositories like Concurrent Versioning System (CVS), and lack fine-grained semantical information (see https://www.cvshome.org/). For example, some approaches offers file and folder changes but give no semantical information about what exactly changed in a system (e.g., classes or methods).

We present briefly three approaches characterizing the work done in the context of history-centered evolution analyses.

2.2.1. Manipulating historical properties: history measurements

The history measurements aim to summarize what happened in the evolution from a particular point of view. Examples of history measurements are: age of an entity, number of changes in an entity, number of authors that changed the system etc.

Ball and Eick [39] developed multiple visualizations for showing changes that appear in the source code. For example, they show what is the percentage of bug fixes and feature addition in files, or which lines were changed recently. Collberg *et al.* use graph-based visualizations to display which parts class hierarchies authors change [40]. They provide a color scale to distinguish between newer and older changes. Xiaomin Wu *et al.* visualize the change log information to provide for an overview of the active places in the system as well as of the authors activity [41]. They display measurements like the number of times an author changed a file, or the date of the last commit. Chuah and Eick present a way to visualize project information through glyphs. Their infobug glyph's parts represent data about software [36]. They use glyphs for viewing project management data (i.e., evolution aspects, programming languages used, and errors found in a software component) and they present time wheel to show the evolution of a given characteristic over time.

Typically, in the literature we find measurements which are very close to the type of information available in the versioning systems. As versioning systems provide textual information like lines of code added/removed, the measurements too only measure the size of the change in lines of code. Even though lines of code can be a good indicator for general overviews, it is not a good indicator when more sensitive information is needed. For example, if 10 lines of code are added in a file, this approach does not distinguish whether the code was added to an existent method, or if several completely new methods were added.

Requirement: *An evolution meta-model should provide detailed information at different levels of abstraction for understanding the changes.*

Requirement: *An evolution meta-model should provide for comparison of property evolutions.*

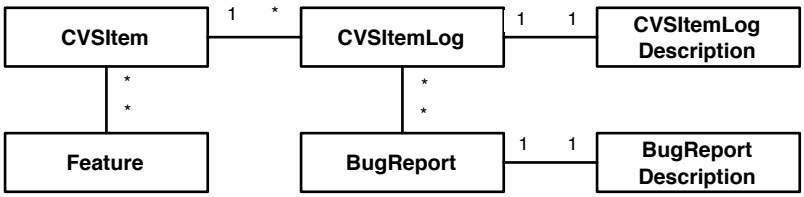

Figure 4. The Release History Meta-Model shows how Features relate to the CVSItem.

2.2.2. Manipulating historical entities: Hipikat and Release History Meta-Model

Fischer *et al.* modeled bug reports in relation to version control system (CVS) items [42]. Figure 4 presents an excerpt of the Release History meta-model. The purpose of this meta-model is to provide a link between the versioning system and the bug reports database. This meta-model recognizes the notion of the history (i.e., CVSItem) which contains multiple versions (i.e., CVSItemLog). The CVSItemLog is related to a Description and to BugReports. Furthermore, it also puts the notion of Feature in relation with the history of an item. The authors used this meta-model to recover features based on the bug reports [43]. These features get associated with a CVSItem.

The main drawback of this meta-model is that the system is represented with only files and folders, and it does not take into consideration the semantical software structure (e.g., classes or methods). Because it gives no information about what exactly changed in a system, this meta-model does not offer support for analyzing the different types of change. Recently, the authors started to investigate how to enrich the Release History Meta-Model with source code information [44].

Čubranić and Murphy bridged information from several sources to form what they call a "group memory"[45]. Čubranić details the meta-model to show how they combined CVS repositories, mails, bug reports and documentation [46].

Draheim and Pekacki presented the meta-model behind Bloof [47]. The meta-model is similar to CVS: a File has several Revisions and each Revision has attached a Developer. They used it for defining several measurements like the Developer cumulative productivity measured in changed LOC per day.

Requirement: *An evolution meta-model should provide for relationships between histories.*

2.2.3. Manipulating historical relationships: historical co-change

Gall *et al.* aimed to detect logical coupling between parts of the system [13] by identifying the parts of the system which change together. They use this information to define a coupling measurement based on the fact that the more times two modules were changed at the same time, the more they were coupled.

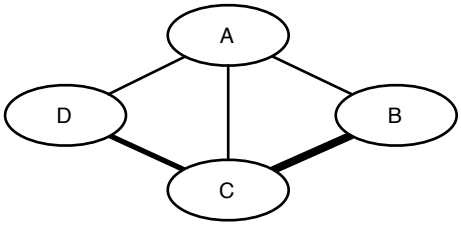

Figure 5. Historical co-change example. Each ellipse represents a module and each edge represents a co-change relationship. The thickness of the edge is given by the number of times the two modules changed together.

Hassan *et al.* analyzed the types of data that are good predictors of change propagation, and came to the conclusion that historical co-change is a better mechanism than structural dependencies like call-graphs [48]. Zimmermann *et al.* defined a measurement of coupling based on co-changes [49].

Zimmermann *et al.* aimed to provide a mechanism to warn developers about the correlation of changes between functions. The authors placed their analysis at the level of entities in the meta-model (e.g., methods) [50]. They presented the problems residing in mining the CVS repositories, but they did not present the meta-model [51].

Similar work was carried out by Ying *et al.* [52]. The authors applied the approach on two large case studies and analyzed the effectiveness of the recommendations. They concluded that although the "precision and recall are not high, recommendations can reveal valuable dependencies that may not be apparent from other existing analyses."

Xing and Stroulia used the fine-grained changes provided by UML Diff to look for class co-evolution [18, 19]. They took the type of changes into account when reasoning, and they distinguish between intentional co-evolution and "maintenance smells" (e.g., Shotgun Surgery and Parallel Inheritance).

Eick *et al.* used the number of files changed in the same time as an one indicator of code decay [5]. They reported on a large case study that changes are more dispersed at the end of the project, which they interpreted as a sign of code decay.

In general, the result of the co-change analysis is that two entities (e.g., files) have a relationship if they were changed together. Gall *et al.* provided a visualization, as in Figure 5, to show how modules changed in the same time [53]. The circles represent modules, and the edges represent the co-change relationship: the thicker the edge, the more times the two modules were changed in the same time. In this representation the structural elements from the last version (i.e., the modules) are linked via a historical relationship (i.e., the co-change relationship).

As in the case of the Evolution Matrix (e.g., where classes were said to be growing), here too there is a conceptual problem from the modeling point of view: co-change actually links the evolution of the entities and not a particular version of the entities. In this case too we

would like to have a reification of the evolution (i.e., history), to be able to relate it to the co-change relationship (see Section 4.5).

Requirement: *An evolution meta-model should provide for relationships between histories.*

2.2.4. Discussion

While in the version-centered analyses, the approach was to present the version information and let the user detect the patterns, in the above examples, the aim is to summarize what happened in the history according to a particular point of view.

For example, an evolution chart displays the versioning data and the user can interpret it in different ways according to the point of view: she can see whether it grows or not, she can see whether it fluctuates or not and so on. As opposed to that, the history measurements encode these points of view and return the values that summarize the evolution. In this case, it is not the reengineer that has to identify the trends or patterns in the history, with the possibility of missing important information. In general, history measurements automate the analysis.

In general, the analyses are influenced by the types of information available. For example, as versioning systems offer information related to the changes of the lines of code, the analyses, too, use addition/deletion of lines code as an indicator of change. While this might be suitable for general overviews, it is not enough for detailed analyses. For example, if we want to detect signs of small fixes, we might look for classes where no method has been added, while only the internal implementation changed.

2.3. Requirements for an evolution meta-model

An evolution meta-model should allow the expression of all of the above analyses and more. Based on the analysis of their underlying meta-models, an evolution meta-model should have the following properties:

Different abstraction and detail levels. The meta-model should provide information at different levels of abstraction such as files, packages, classes, methods for each analyzed version. For example, CVS meta-model offers information about how source code changed (e.g., addition, removals of lines of code), but it does not offer information about additions or removals of methods in classes.

The meta-model should support the expression of detailed information about the structural entity. For example, knowing the authors that changed the classes is an important information for understanding evolution of code ownership.

Comparison of property evolutions. The meta-model should offer means to easily quantify and compare how different entities evolved with respect to a certain property. For example, we must be able to compare the evolution of number of methods in classes, just like we can compare the number of methods in classes. For that, we need a way to quantify how the number of methods evolve and afterwards we need to associate such a property with an entity.

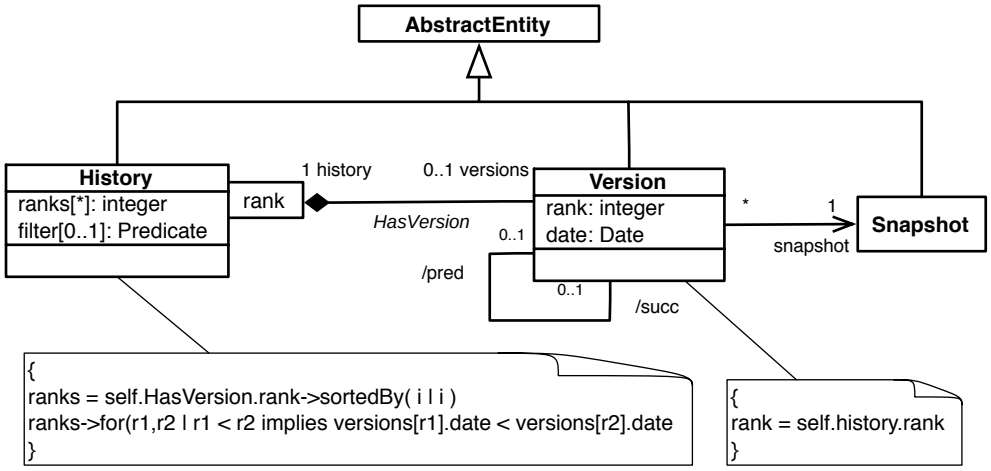

Figure 6. Details of the relationship between the History, the Version and the Snapshot. A History has a container of Versions. A Version wraps a Snapshot and adds evolution specific queries.

Combination of different property evolutions. The meta-model should allow for an analysis to be based on the evolution of different properties. Just like we reason about multiple structural properties, we want to be able to reason about how these properties have evolved. For example, when a class has only a few methods, but has a large number of lines of code, we might conclude it should be refactored. In the same line, adding or removing the lines of code in a class while preserving the methods we might conclude the change was a bug-fix.

Selectability. The analysis should be applicable on any group of versions (i.e., we should be able to select any period in the history).

Navigation. The meta-model should provide relations between histories to allow for navigation. For example, we should be able to ask our model which methods ever existed in a particular class, or which classes in a particular package have been created in the last period of time.

3. HISMO: MODELING HISTORY AS A FIRST CLASS ENTITY

In this section we introduce Hismo, our solution of modeling history to support software evolution analysis: explicitly model history as a ordered set of versions. The core of Hismo is based on three entities: *History*, *Version* and *Snapshot*. Figure 6 shows the relationships between these entities in a UML 2.0 diagram:

Snapshot. This entity is a placeholder that represents the entities whose evolution is studied i.e., file, package, class, methods or any source code artifacts. The particular entities are to be sub-typed from Snapshot as shown in Figure 8.

History. A History holds a set of Versions. The relationship between History and Version is depicted with a qualified composition which depicts that in a History, each Version is uniquely identified by a rank. From a History we can obtain a sub-History by applying a filter predicate on the set of versions.

Version. A Version adds the notion of time to a Snapshot by relating the Snapshot to the History. A Version is identified by a time-stamp and it knows the History it is part of. A Version can exist in only one History. Based on its rank in the History, Version has zero or one predecessor and zero or one successor.

In Hismo, we add time information as a layer on top of the snapshot information. As such, the snapshot data can exist without any reference to history but can still be manipulated in the context of software evolution. Because of this, Hismo can be built on top of *any* snapshot meta-model without interfering with the existing meta-model. There are many meta-models describing structural information, and many analyses are built on this meta-models. With our approach of we can reuse the analyses at structural level and include them in the evolution analysis.

History, Version and Snapshot are abstract and generic entities, and as such, the core of Hismo is not tied to any meta-model. These concepts are generic and can be applied to any kind of entities such as packages, classes, methods or any entity related to the system that we want to study as shown by Figure 8.

3.1. Building Hismo based on a snapshot meta-model

In this section we show how to apply the generic concepts of History, Version and Snapshot to specific snapshot meta-models.

We start by taking a detailed look at Hismo applied to Packages and Classes (see Figure 7). There is a parallelism between the version entities and the history entities: Each version entity has a correspondent history entity. Also, the relationship at version level (e.g., a Package has more Classes) has a correspondent at the history level (e.g., a PackageHistory has more ClassHistories).

Figure 8 shows an overview of the history meta-model based on a larger source-code meta-model. Here we use FAMIX, a language independent source code meta-model [54]. The details of the full meta-model are similar to the one in Figure 7.

The snapshot entities (e.g., Method) are wrapped by a Version correspondent (e.g., MethodVersion) and the Versions are contained in a History (e.g., MethodHistory). A History does not have direct relation with a Snapshot entity, but through a Version wrapper as shown in Figure 6. We create Versions as wrappers for SnapshotEntities because in a Version we store the relationship to the History: a Version is aware of the containing History and of its position in the History (i.e., it knows the predecessor and the successor). Thus, we are able to compute

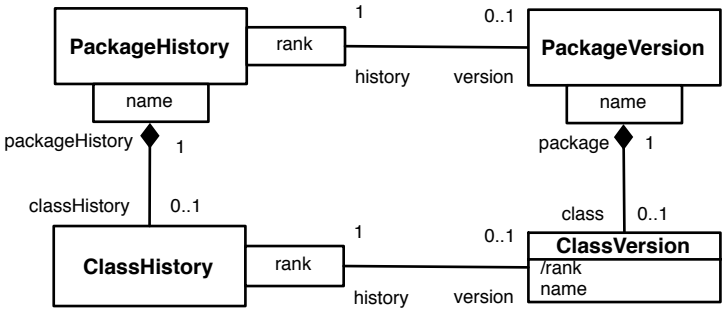

Figure 7. Hismo applied to Packages and Classes.

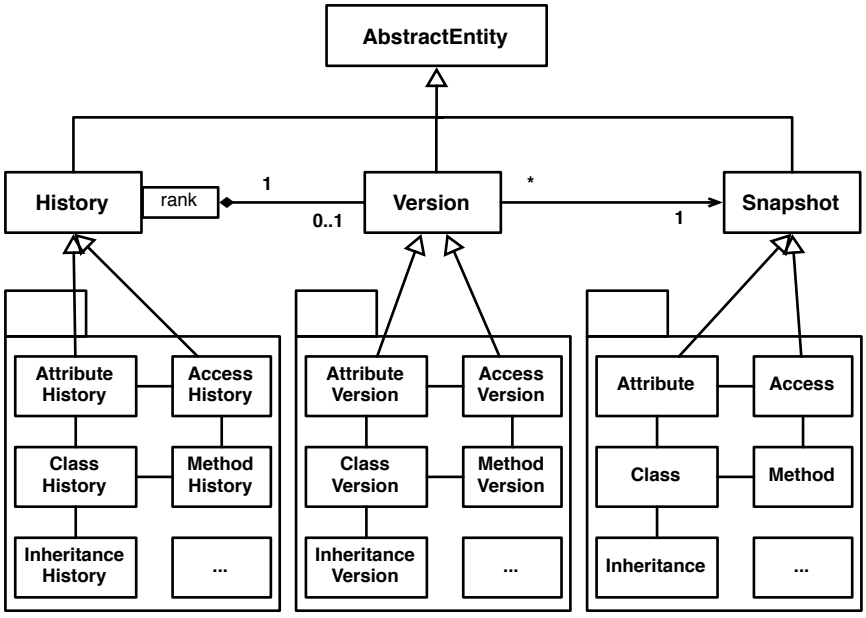

Figure 8. An excerpt of Hismo as applied to FAMIX, and its relation with a snapshot meta-model: Every snapshot entity is wrapped by a corresponding Version and a set of Versions form a History. We did not represent all the inheritance relationships to not affect the readability of the picture.

properties for a particular Version in the context of the History. For example, having a Version we can navigate to the previous or the next Version.

A problem raised in the literature is that of what we call *entity identity* (it can also be found as *origin analysis* [20, 55, 56]). The most common way to recover the identity is by the name of the entity, that is if we have two entities with the same name and the same type in two versions, then they are considered to be two versions of the same entity. Of course, such approaches miss refactorings like renaming or moving. Different approaches have been proposed to solve this problem: using information retrieval techniques [20], using string matching or entities fingerprints [55, 56].

In our definition, the history is a set of versions, therefore, it also encapsulates the entity identity. We did not specify the algorithm to be used when determining entity, because it is the responsibility of the implementation to determine how the identity is defined. For example, it is possible to first determine the histories based on names and then detect renaming refactorings and merge the histories that are detected as being renamed.

3.2. Mapping Hismo to the Evolution Matrix

In this section we describe how Hismo maps to the Evolution Matrix (see Figure 9). In the upper part of the figure we represent Hismo applied to Packages and Classes where a package contains several classes, while in the lower part we show two Evolution Matrices. As described by Figure 2 (Section 2.1.3), a row represents the evolution of an entity, here a class and a column all the entities of one version i.e., a package. Therefore in Figure 9 each cell in the matrix represents a ClassVersion and each column represents a PackageVersion.

In Hismo a *history* is a sequence of *versions*. Thus, each line in an Evolution Matrix represents a ClassHistory (left matrix). Moreover, the whole matrix is actually a line formed by PackageVersions (right matrix), which means that the whole matrix can be seen as a PackageHistory (left matrix).

In the upper part we also represent the relations we have between the entities. On the right part we show that a PackageVersion *has* multiple ClassVersions, while on the left side we show that in a PackageHistory *there are* multiple ClassHistories.

4. USING HISMO FOR SOFTWARE EVOLUTION ANALYSIS

In this section we show examples how Hismo meets the requirements we described in the previous section. These examples serve as validation of the expressiveness of our meta-model, as each of them exercise different characteristics of Hismo.

First, we introduce some history measurements (Section 4.1) and then we use them in three reverse engineering applications:

- Definition of complex historical measurements [10] (Section 4.2);
- Definition of automatic queries which combine different evolution characteristics with version information to improve the detection of design flaws [11] (Section 4.3);

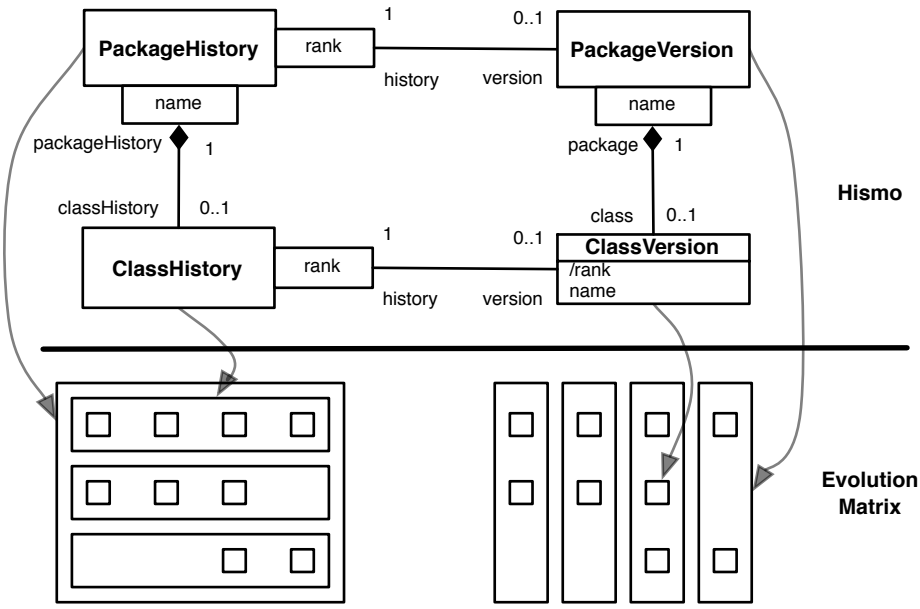

Figure 9. Mapping Hismo to the Evolution Matrix. Each cell in the Evolution Matrix represents a version of a class. Each column represents the version of a package. Each line in the Evolution Matrix represents a history. The entire matrix displays the package history.

- Visualization of different historical measurements to determine patterns of evolution and to determine correlations between different kinds of changes [12] (Section 4.4).

Furthermore, in Section 4.5 we show how Hismo can be used for co-change analysis.

4.1. History measurements and properties

As discussed in Section 2.2.1, history measurements quantify the changes in the history according to a particular interest. The benefit of the historical measurements is that we can understand what happened with an entity *without* a detailed look at each version – i.e., the measurements summarize changes into numbers which are assigned to the corresponding histories.

The problem with most of the existing measurements is that they do not take into account the semantical meaning of the system structure, but they usually rely on primary data like lines of code, files and folders. Such measurements are of limited use when we need fine grained information.

Figure 10 gives an example how we can use the detailed information in Hismo to define historical measurements:

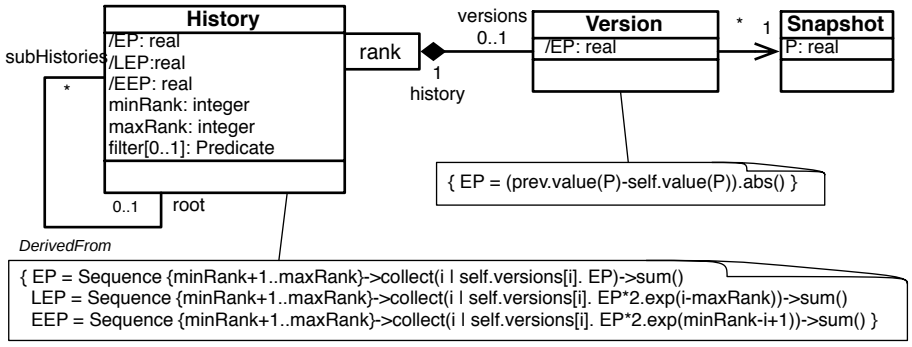

Figure 10. Examples of history measurement definitions. The Snapshot has structural numerical properties denoted here with P (e.g., P can be the number of methods in a class). Based on these properties we can define history measurements (e.g., in this case we show EP, LEP and EEP).

- *Evolution of a property P (*EP*)* – this measurement is defined as the sum of the absolute difference of P in subsequent versions. This measurement can be used as an overall indicator of change.
- *Latest Evolution of P (*LEP*)* – while *EP* treats each change the same, with *LEP* we focus on the latest changes by a weighting function $2^{i-maxRank}$ which decreases the importance of a change as the version i in which it occurs is more distant from the latest considered version $maxRank$.
- *Earliest Evolution of P (*EEP*)* – it is similar to *LEP*, only that it emphasizes the early changes by a weighting function $2^{minRank-i+1}$ which decreases the importance of a change as the version i in which it occurs is more distant from the first considered version $minRank$.

Given a History we can obtain a sub-History based on a filtering predicate applied on the versions. Therefore, whichever properties we can compute on the History, we can also compute on the sub-History.

In Figure 11 we show an example of applying the defined history measurements to 5 histories of 5 versions each.

- During the displayed history of D (5 versions) P remained 2. That is the reason why all three history measurements were 0.
- Throughout the histories of class A, of class B and of class E the P property was changed the same as shown by the Evolution of P (EP = 7). The Latest and the Earliest Evolution of P (LEP and EEP) values differ for the three class histories which means that (i) the changes are more recent in the history of class B (ii) the changes happened in the early past in the history of class E and (iii) in the history of class A the changes were scattered through the history more evenly.

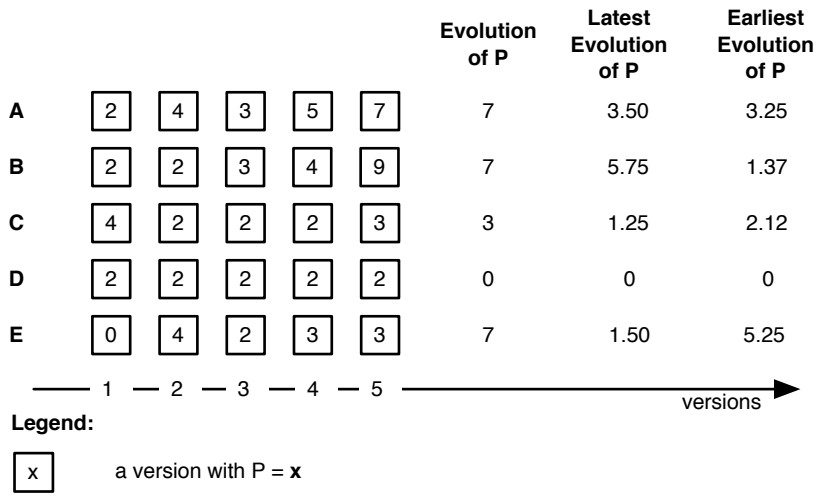

Figure 11. Example of history measurements.

- The histories of class C and E have almost the same LEP value, because of the similar amount of changes in their recent history. The EP values differ heavily because class E was changed more throughout its history than class C.

The above measurements are depend on the P property. For example, P can be the number of methods of a class (NOM), or the number of lines of code of a method (LOC). As a consequence, in the case of EP we talk about ENOM, when P is NOM, or about ELOC when P is LOC.

In a similar fashion, we define other measurements. Here is a non-exhaustive list:

- *Age* – It counts the number of versions in the history.
- *Additions of P / Removals of P* – These measurements sum the additions or removals of a property P. Additions are a sign of increase in functionality, while removals are a sign of refactoring.
- *Number of Changes of P* – It counts in how many versions the property P changed with respect to the previous version.
- *Stability of P* – It divides the Number of Changes of P by the number of versions - 1 (i.e., the number of versions in which P could have changed).
- *History Maximum / Minimum / Average* – This measurements the maximum, minimum or the average value of P over the versions.
- *Persistence of a version Condition* – It counts the number of versions in which the Condition is true over the total number of versions.

Not only measurements can be defined as historical properties. Here are some examples of boolean properties:

- *Persistent* – A persistent entity is an entity that was present in all versions of the system.
- *Day-fly* – Day-fly denotes a history with only one version.
- *Removed* – A history is removed if its last version is not part of the system history's last version

4.2. Yesterday's Weather: selecting histories and combining property evolutions

The measurements defined above are used to define another measurement: Yesterday's Weather (YW) [10]. The YW measurement represents the retrospective empirical observation of the phenomenon that at least one of the classes which were heavily changed in the recent history is also among the most changed classes in the near future.

The approach consists of identifying, for each version of a subject system, the classes that were changed the most in the recent history and in checking if these are also among the most changed classes in the successive versions. The YW value is given by the number of versions in which this assumption holds divided by the total number of analyzed versions. If YW raises a high value, we say it is useful to start reengineering from the classes which changed the most in the recent past, because there is a high chance that they will also be among the most changed in the near future.

Figure 12 shows the OCL code for computing YW for a SystemHistory. The code reveals several features of Hismo:

- We navigate the meta-model by asking a SystemHistory for all the ClassHistories (self.classHistories).
- classHistories->selectTopLENOMFromVersions(minRank, versionRank-1) returns the class histories that are in the top of LENOM (Latest Evolution of Number of Methods) in the period between the first version (minRank) and the version before the wanted version (versionRank-1). That is, it returns the classes that were changed the most in the recent history. This predicate implies applying a historical measurement (i.e., LENOM) on a selection of a history, and then ordering the histories according to the results of the measurement.
- Similarly, classHistories->selectTopEENOMFromVersions(versionRank, maxRank) returns the class histories that are the most changed in the early history between the wanted version (versionRank) and the last version (maxRank).
- The result of versionYW is a boolean showing if the intersection (intersectWith) of the past changed class histories (yesterdayTopHistories) and the future changed class histories (todayTopHistories) is not empty. This simple intersection is possible because the yesterdayTopHistories and todayTopHistories are subsets of all classHistories.

```
context SystemHistory

/* returns true if the YW assumption holds for a given version versionRank */
derive versionYW(versionRank):
  yesterdayTopHistories = self.classHistories->selectTopLENOMFromVersions(minRank, versionRank).
  todayTopHistories = self.classHistories->selectTopEENOMFromVersions(versionRank, maxRank).
  yesterdayTopHistories->intersectWith(todayTopHistories)->isNotEmpty().

/* answers the number of versions in which the assumption holds
   divided by the total number of analyzed versions*/
derive overallYW:
  ywVersionResults = Sequence(minRank+2..maxRank-1)->collect(i | self.versionYW(i).
  ywVersionResults->sum() / (maxRank-minRank-2)
```

Figure 12. The OCL expression for computing Yesterday's Weather.

4.3. History-based detection strategies: combining snapshot properties with historical properties

Another usage of history measurements was proposed for improving design flaws detection by taking the evolution into account [11]. In particular, the work shows how the detection of DataClasses and GodClasses based on structural measurements can be improved by taking into account information such as the stability of the entity or the persistence of the flaw during the lifetime of the entity.

For example, GodClasses are defined as classes that tend to centralize the intelligence of the system [57]. Marinescu defined measurements-based expressions to detect GodClasses [58, 59].

Originally, the detection strategies only took into account structural measurements. We used Hismo to extend the detection strategies with the time dimension. For example, we use the historical information to qualify GodClasses as being harmless if they were stable for a large part of their history, because being stable means those classes were not a maintainability problem in the past (e.g., 95%), and that means that they are likely to not be a problem in the future also. Figure 13 shows the expression we used.

The code shows the predicate isHarmlessGodClass defined in the context of a ClassHistory. The predicate is an example of how we can use in a single expression both historical information (i.e., stability of number of methods) and structural information (i.e., GodClass characteristic in the last version) to build the reasoning.

4.4. Characterizing hierarchies evolution: navigating history, and combining historical properties

Based on Hismo, a visualization has been proposed to detect patterns of hierarchy evolution [12]. The visualization is based on the polymetric view principle [60]. A polymetric view is a

```
context ClassHistory

/* returns true if the ClassHistory is a GodClass in the last version and
   it did not change the number of methods in more then 95% of the versions */
derive isHarmlessGodClass: (self.versions->last().isGodClass()) &
                           (self.stabilityOfNOM > 0.95)
```

Figure 13. The OCL expression for detecting stable GodClasses.

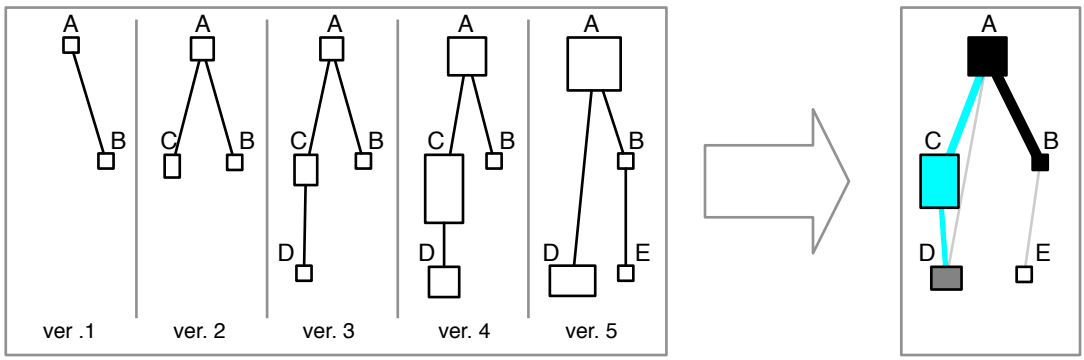

Figure 14. Class Hierarchies Evolution View (right hand side) represents class hierarchy histories. Node width = Evolution of Number of Methods (ENOM); Node height = Evolution of Number of Statements (ENOS); Node color = Age; Edge width = Age; Edge color = Age; Cyan node or edge = Removed history.

visualization technique which displays entities as boxes and relationships as edges, where the size and color of the boxes and edges are given by metric results of the represented entities and relationships.

Originally, polymetric views were only used for analyzing structural information (e.g., the hierarchy of classes). Figure 14 shows an example of how the evolution over 5 versions of a hierarchy (left side) is summarized in one polymetric view (right side). On the left hand side we represent the hierarchy structure in each version – i.e., classes as nodes and inheritance relationships as edges. On the right hand side we display ClassHistories as nodes and InheritanceHistories as edges. The color of each node represents the age of the ClassHistory: the darker the node the older the ClassHistory. The size of the node denotes how much it was changed: the larger the node, the more the ClassHistory was changed. Both the thickness and the color of the edge represent the age of the InheritanceHistory. Furthermore, the cyan color denotes removed histories.

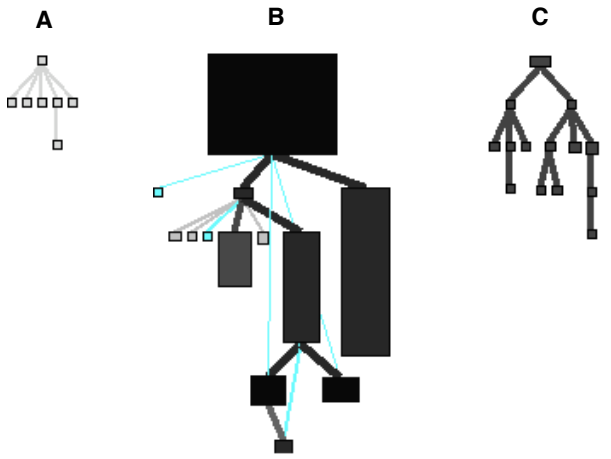

Figure 15. Examples of class hierarchies evolution reavealing different evolution patterns. A is a young hierarchy. B is unstable. C is an old and stable hierarchy.

Figure 15 shows the visualization applied on the history of three class hierarchies (denoted with A, B and C). The nodes represent class histories and the edges inheritance histories. Both the nodes and the edges are annotated with historical measurements. The width of the nodes is given by the Evolution of Number of Methods (ENOM), the height is given by the Evolution of Number of Statements (ENOS) and the color is given by the age of the class. The width and the color of the edges is given by the age of the inheritance. The cyan color denotes remove classes or inheritances.

From the visualization we can characterize the evolution of class hierarchies. In our example, hierarchy A has small and white nodes and thin edges which means that it is newly introduced in the system. Hierarchy C, is an old one and stable because the nodes are small and the inheritance relationships never changed. In the B hierarchy, there are nodes with different shapes and colors denoting there were many changes in this hierarchy. Also, we see cyan edges which denotes removed inheritance due to introduction of super classes in the middle of the hierarchy.

4.5. Identifying co-change patterns: manipulating historical relationships

Figure 16 shows how to use Hismo in co-change analysis. On the bottom part we represent the example from Figure 5: On the bottom-left we show 6 versions of 4 modules (A, B, C, D) and how they changed from one version to the other (marked with gray). On the bottom-right we show the historical representation. The resulting picture is the same as in Figure 5 only the

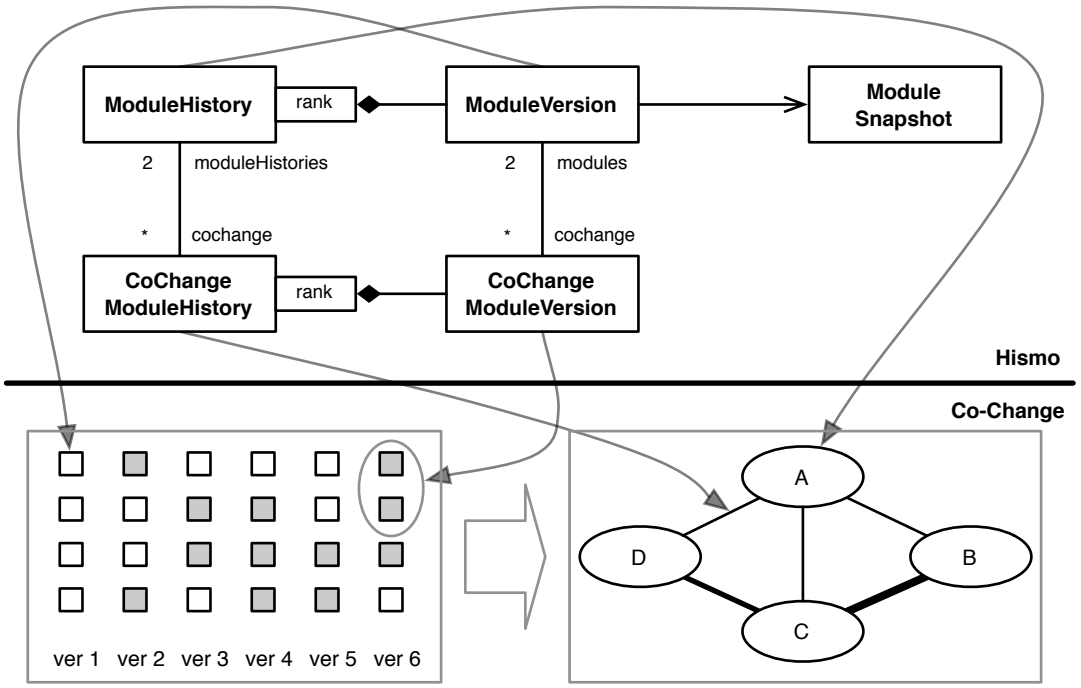

Figure 16. Using Hismo for co-change analysis. On the bottom-left side, we show 6 versions of 4 modules: a grayed box represent a module that has been changed, while a white one represents a module that was not changed. On the bottom-right side, we show the result of the evolution of the 4 modules as in Figure 5.

meta-model is different. We no longer represent the Modules as ellipses, but ModuleHistories, and the co-change is an explicit historical relationship (CoChangeModuleHistory).

We used this meta-model to develop an analysis of detecting co-change patterns using Formal Concept Analysis [8]. Formal Concept Analysis is a generic technique that takes as an input entities and properties and returns concepts, where each concept contains a set of entities and a set of properties common to the entities. We aimed to detect parallel inheritance. For that we used as entities, ClassHistories, and we used as the i^{th} property the fact that a ClassHistory changed in version i.

5. DISCUSSION: HOW HISMO SATISFIES THE REQUIREMENTS

The above applications show that Hismo satisfies the desired requirements:

Different abstraction and detail levels. Hismo takes into account the structure of the system. We gave examples of history measurements which take into account different semantics of change (e.g., changes in number of methods, number of statements) on different entities (e.g., packages, classes, methods). As we show Hismo can be applied on any software entities which can play the role of a Snapshot Entity.

Furthermore, having detailed information about each version, we can also define the version properties (e.g., number of methods) in terms of history (e.g., number of methods in version i). Thus, all things we can find out from one version can be found out having the entire history. This allows for combining structural data with evolution data in the same expression. For example, we can detect the harmless GodClasses by detecting those that were stable.

Evolution comparison. History describes the evolution. History measurements are a way to quantify the changes and they allow for the comparison of different entity evolutions. For example, the Evolution of Number of Methods lets us assess which classes changed more in terms of added or removed methods.

Combination of different property evolutions. In the above examples we showed how we can combine different property evolutions with each other (see Figure 15) or with structural information (see Section 4.3).

Selectability. Figure 10 shows that given a history we can filter it to obtain a sub-history. As the defined analyses are applicable on a history, and a selection of a history is another history, the analyses described in the previous section can be applied on any selection.

Navigation. The Figure 9 shows how based on the structural relationship "a Package has Classes" we can build the relationship "a PackageHistory has ClassHistories". This can be generalized to any structural relationship and thus, at the history level we can ask a PackageHistory to return all ClassHistories – i.e., all Classes that ever existed in that Package.

6. IMPLEMENTATION: MOOSE AND VAN

From our experience, when it comes to large systems, it is not enough to have prefabricated report generators, but it is crucial to have the ability to query and navigate the system under investigation. Furthermore, the more data we need to analyze, the more analyses we need to apply, and different analyses are implemented by different people in different tools. Thus, we focus on dynamism and integration of tools.

In our implementation Hismo is built on top of FAMIX, a language independent meta-model [54]. As an implementation platform we use the Moose reengineering environment [14]. The architecture of Moose is schematically presented in Figure 17. Moose has a repository that can store multiple models, thus providing the necessary infrastructure for holding and managing multiple versions.

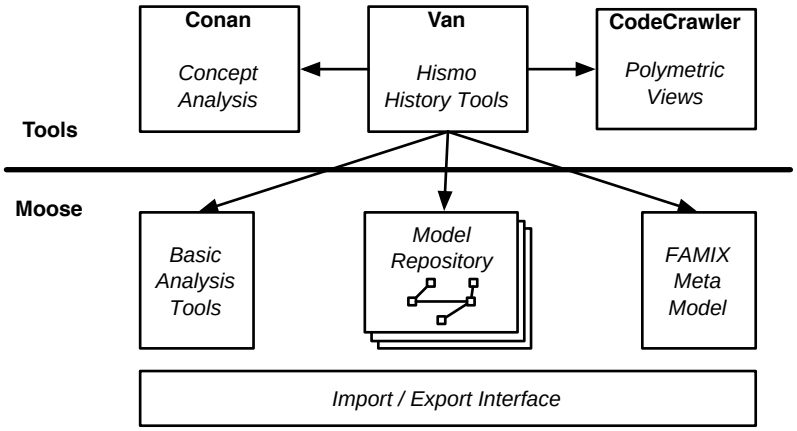

Figure 17. Moose is an extensible reengineering environment. Different tools have been developed on top of it (e.g., Van is our history analysis tool). The tools layer can use and extend anything in the environment including the meta-model. The model repository can store multiple models in the same time. Different languages can be loaded either directly or via intermediate data formats.

Hismo is the foundation of Van, an evolution analysis tool built on top of Moose. Moose is designed to be extensible from any perspective including the meta-model. Van extends the meta-model with the notion of history, and when Van is loaded the entire environment is history-aware. Moose allows for attaching meta-annotations to the entities in the meta-model. For example, among the annotations is the menu: each entity in the meta-model has an interaction description which can be interpreted as a menu.

Van uses CodeCrawler for visualizing polymetric views [61]. In Section 4.4 we showed how we used polymetric views for visualizing hierarchies evolution. In our implementation it was straight forward to combine Van with CodeCrawler due to the meta-model, because CodeCrawler can visualize models by interpreting their meta-models.

We also uses ConAn (see Section 4.5), a Formal Concept Analysis tool to detect co-change patterns [62]. In this case too, the bridge was straight forward, because ConAn manipulates entities and properties, and we provided with histories as entities and "changed in version i" as properties.

Figure 18 shows screenshots with three windows: a Hierarchy Evolution Complexity View in CodeCrawler; a History Inspector showing the details of the selected ClassHistory from the view, and an Evolution Chart of how the number of methods evolved in the selected ClassHistory. The figure stresses the dynamic aspect of the environment by showing how the user can interact with the objects under analysis using contextual menus. When we select an object, we can invoke the attached menu based on its entity description. Furthermore, each tool can extend the menu with its own functionality.

By making history an explicit entity in the meta-model, we can attach the any annotation to it, including the menu.

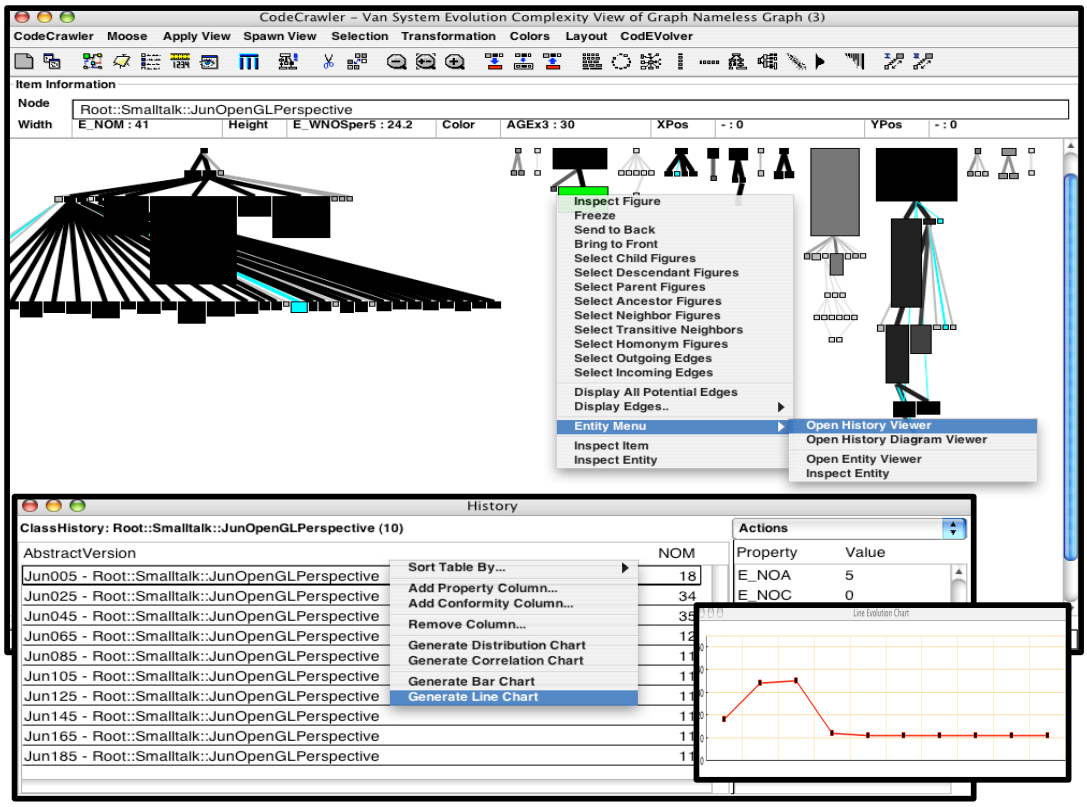

Figure 18. Screenshots with Van and CodeCrawler. The entities are interacted with via contextual menus.

7. CONCLUSIONS

Understanding software evolution is important as evolution holds information that can be used in various analyses like: reverse engineering, prediction, change impact, or in developing general laws of evolution.

We have reviewed various approaches used to understand the software evolution. These approaches typically focus on only some traits of the evolution, and most of them do not rely on an explicit meta-model. Of course, all analyses developed so far had to have a way of representing the data to be able to be implemented. Nevertheless, to be able to compare the results and combine the analyses, we need to make explicit the underlying meta-model.

Based on our investigation of evolution analyses we have gathered requirements for a general evolution meta-model: (1) different detail and abstraction levels, (2) property

evolutions comparison, (3) combination of different property evolutions, (4) selectability, and (5) navigation.

We introduced Hismo, our evolution meta-model which has in its center three explicit notions: History, Version and Snapshot. In short, a Version places the Snapshot into the History. Based on these notions we can construct the historical meta-model based on any snapshot meta-model. In this way, we do not need to change the existing meta-models to analyze the evolution, and like that we can reuse the analyses at structural level and extend them in the context of evolution analysis.

We implemented Hismo and several analyses in Van. We briefly sketched the implementation to show the benefits of having an explicit meta-model when building and combining analysis tools.

In this article, we modeled history as a sequence of versions which implies a linear version alignment. In the future, we would like to investigate the implications of modeling history as a partially ordered set of versions to represent time as a graph.

APPENDIX A: GLOSSARY OF TERMS

For entity, we use the definition as found in the Webster Dictionary:

> An *entity* is something that has separate and distinct existence in objective or conceptual reality.

For the general terms of model and meta-model we use the following definitions:

> A *model* is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system [7].

> A *meta-model* is a specification model for a class of systems under study where each system under study in the class is itself a valid model expressed in a certain modeling language [6].

Throughout the article we use evolution specific terms: version, evolution, and history. We define these terms as follows:

> A *snapshot* is the structure of an entity at a particular moment in time.

> A *version* is a snapshot of an entity placed in the context of time.

In the context of CVS, version is denoted by revision.

> The *evolution* is the process that leads from one version of an entity to another.

> A *history* is the reification which encapsulates the knowledge about evolution and version information.

Following these definitions, we say that we use the history of a system to understand its evolution. Furthermore, the evolution refers to all changes from a version of an entity to another. Sometimes, however, we need to refer to only the change of a particular property of that entity. That is why we define:

Property evolution denotes how a particular property evolved in an entity.

Historical property denotes a characteristic of a history.

For example, the age of a file in CVS is a historical property.
We use different terms for different types of models and meta-models:

A *snapshot meta-model*, or a *structural meta-model* is a meta-model which describes the structure of a class of systems at a certain moment in time.

Examples of snapshot meta-models are UML or FAMIX.

An *evolution model* is a simplified view on the evolution of a system.

Examples of evolution models include the date sequence of each release, a chart showing team allocation over time for a given set of modules, the modifications performed etc.

An *evolution meta-model* is a meta-model which describes a family of evolution models.

For instance, in each versioning systems there is an evolution meta-model that specifies which kind of information is kept about evolution.

A *history meta-model* is an evolution meta-model which models history as a first class entity.

ACKNOWLEDGEMENTS

We would like to thank the reviewers who gave valuable feedback. In particular we would like to thank Orla Greevy, Jean-Marie Favre and Jean-Marc Jezequel.

REFERENCES

1. Rochkind M. The source code control system. *IEEE Transactions on Software Engineering*, 1975; **1**(4):364–370.
2. Hunt J, McIlroy D. An algorithm for differential file comparison. *Technical Report CSTR 41*. Bell Laboratories: Murray Hill NJ, 1976; 9 pp.
3. Lehman M, Belady L. *Program Evolution: Processes of Software Change*. London Academic Press: London, 1985; 538 pp.
4. Parnas D. L. Software aging. *Proceedings 16th International Conference on Software Engineering (ICSE '94)*. IEEE Computer Society: Los Alamitos CA, 1994; 279–287.
5. Eick S, Graves T, Karr A, Marron J, Mockus A. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 2001; **27**(1):1–12.
6. Seidewitz E. What models mean. *IEEE Software*, 2003; **20**(5):26–32.
7. Bézivin J, Gerbé O. Towards a precise definition of the OMG/MDA framework. *Proceedings Automated Software Engineering (ASE 2001)*. IEEE Computer Society: Los Alamitos CA, 2001; 273–282.
8. Gîrba T. Modeling history to understand software evolution. *Doctoral dissertation*. University of Berne: Berne, 2005; 164 pp.
9. Ducasse S, Gîrba T, Favre J.-M. Modeling software evolution by treating history as a first class entity. *Proceedings Workshop on Software Evolution Through Transformation (SETra 2004)*. Elsevier: Amsterdam, 2004; 75–86.
10. Gîrba T, Ducasse S, Lanza M. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 40–49.
11. Raţiu D, Ducasse S, Gîrba T, Marinescu R. Using history information to improve design flaws detection. *Proceedings Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*. IEEE Computer Society: Los Alamitos CA, 2004; 223–232.
12. Gîrba T, Lanza M, Ducasse S. Characterizing the evolution of class hierarchies. *Proceedings Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*. IEEE Computer Society: Los Alamitos CA, 2005; 2–11.
13. Gall H, Hajek K, Jazayeri M. Detection of logical coupling based on product release history. *Proceedings International Conference on Software Maintenance (ICSM '98)*. IEEE Computer Society Press: Los Alamitos CA, 1998; 190–198.
14. Ducasse S, Gîrba T, Lanza M, Demeyer S. Moose: a collaborative and extensible reengineering Environment. *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series. Franco Angeli: Milano, 2005; 55–71.
15. Nierstrasz O, Ducasse S, Gîrba T. The story of Moose: an agile reengineering environment. *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*. ACM Press: New York NY, 2005; 1–10.
16. Demeyer S, Ducasse S, Nierstrasz O. Finding refactorings via change metrics. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM Press: New York NY, 2000; 166–178.
17. Xing Z, Stroulia E. Understanding phases and styles of object-oriented systems' evolution. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 242–251.
18. Xing Z, Stroulia E. Data-mining in support of detecting class co-evolution. *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE 2004)*. ACM Press: New York NY, 2004; 123–128.
19. Xing Z, Stroulia E. Understanding class evolution in object-oriented software. *Proceedings 12th IEEE International Workshop on Program Comprehension (IWPC'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 34–43.
20. Antoniol G, Di Penta M. An automatic approach to identify class evolution discontinuities. *Proceedings IEEE International Workshop on Principles of Software Evolution (IWPSE 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 31–40.
21. Lehman M, Perry D, Ramil J. Implications of evolution metrics on software maintenance. *Proceedings IEEE International Conference on Software Maintenance (ICSM'98)*. IEEE Computer Society Press: Los Alamitos CA, 1998; 208–217.
22. Lehman M. Laws of software evolution revisited. *European Workshop on Software Process Technology*. Springer: Berlin, 1996; 108–124.

23. Lehman M, Perry D, Ramil J, Turski W, Wernick P. Metrics and laws of software evolution – the nineties view. *Proceedings Fourth International Software Metrics Symposium (METRICS'97)*. IEEE Computer Society Press: Los Alamitos CA, 1997; 20–32.
24. Ramil J, Lehman M. Defining and applying metrics in the context of continuing software evolution. *Proceedings of the 7th International Symposium on Software Metrics (METRICS '01)*. IEEE Computer Society Press: Los Alamitos CA, 2001; 199–209.
25. Gall H, Jazayeri M, Klösch R, Trausmuth G. Software evolution observations based on product release history. *Proceedings International Conference on Software Maintenance (ICSM'97)*. IEEE Computer Society Press: Los Alamitos CA, 1997; 160–166.
26. Godfrey M, Tu Q. Evolution in open source software: A case study. *Proceedings International Conference on Software Maintenance (ICSM 2000)*. IEEE Computer Society Press: Los Alamitos CA, 2000; 131–142.
27. Capiluppi A. Models for the evolution of OS projects. *Proceedings International Conference on Software Maintenance (ICSM 2003)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 65–74.
28. Capiluppi A, Morisio M, Lago P. Evolution of understandability in OSS projects. *Proceedings 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 58–66.
29. Lanza M, Ducasse S. Understanding software evolution using a combination of software visualization and software metrics. *Proceedings of Langages et Modèles à Objets (LMO 2002)*. Lavoisier: Paris, 2002; 135–149.
30. Burd E, Munro M. An initial approach towards measuring and characterizing software evolution. *Proceedings of the Working Conference on Reverse Engineering, (WCRE 1999)*. IEEE Computer Society Press: Los Alamitos CA, 1999; 168–174.
31. Taylor C, Munro M. Revision towers. *Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis*. IEEE Computer Society: Los Alamitos CA, 2002; 43–50.
32. Van Rysselberghe F, Demeyer S. Studying software evolution information by visualizing the change history. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 328–337.
33. Wu J, Holt R, Hassan A. Exploring software evolution using spectrographs. *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 80–89.
34. Jazayeri M. On architectural stability and evolution. *Reliable Software Technologies-Ada-Europe 2002*. Springer Verlag: Berlin, 2002; 13–23.
35. Eick S, Graves T, Karr A, Mockus A, Schuster P. Visualizing software changes. *IEEE Transactions on Software Engineering*, 2002; **28**(4):396–412.
36. Chuah M. C, Eick S. G. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 1998; **18**(4):24–29.
37. Holt R, Pak J. GASE: Visualizing software evolution-in-the-large. *Proceedings of Working Conference on Reverse Engineering (WCRE 1996)*. IEEE Computer Society Press: Los Alamitos CA, 1996; 163–167.
38. Gulla B. Improved maintenance support by multi-version visualizations. *Proceedings Conference on Software Maintenance (ICSM 1992)*. IEEE Computer Society Press: Los Alamitos CA, 1992; 376–383.
39. Ball T, Eick S. Software visualization in the large. *IEEE Computer*, 1996; **29**(4):33–43.
40. Collberg C, Kobourov S, Nagra J, Pitts J, Wampler K. A system for graph-based visualization of the evolution of software. *Proceedings of the 2003 ACM Symposium on Software Visualization*. ACM Press: New York NY, 2003; 77–86.
41. Wu X, Murray A, Storey M.-A, Lintern R. A reverse engineering approach to support software maintenance: Version control knowledge extraction. *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 90–99.
42. Fischer M, Pinzger M, Gall H. Populating a release history database from version control and bug tracking systems. *Proceedings International Conference on Software Maintenance (ICSM 2003)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 23–32.
43. Fischer M, Pinzger M, Gall H. Analyzing and relating bug report data for feature tracking. *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 90–99.
44. Antoniol G, Di Penta M, Gall H, Pinzger M. Towards the integration of versioning systems, bug reports and source code meta-models. *Proceedings Workshop on Software Evolution Through Transformation (SETra 2004)*. Elsevier: Amsterdam, 2004; 83–94.
45. Čubranić D, Murphy G. Hipikat: Recommending pertinent software development artifacts. *Proceedings 25th International Conference on Software Engineering (ICSE 2003)*. ACM Press: New York NY, 2003;

408–418.
46. Čubranić D. Project history as a group memory: Learning from the past. *Doctoral dissertation*. University of British Columbia: Vancouver BC, 2004; 149 pp.
47. Draheim D, Pekacki L. Process-centric analytical processing of version control data. *International Workshop on Principles of Software Evolution (IWPSE 2003)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 131–136.
48. Hassan A, Holt R. Predicting change propagation in software systems. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 284–293.
49. Zimmermann T, Diehl S, Zeller A. How history justifies system architecture (or not). *6th International Workshop on Principles of Software Evolution (IWPSE 2003)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 73–83.
50. Zimmermann T, Weißgerber P, Diehl S, Zeller A. Mining version histories to guide software changes. *26th International Conference on Software Engineering (ICSE 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 563–572.
51. Zimmermann T, Weißgerber P. Preprocessing CVS data for fine-grained analysis. *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 2–6.
52. Ying A, Murphy G, Ng R, Chu-Carroll M. Predicting source code changes by mining change history. *Transactions on Software Engineering*, 2004; **30**(9):573–586.
53. Gall H, Jazayeri M, Krajewski J. CVS release history data for detecting logical couplings. *International Workshop on Principles of Software Evolution (IWPSE 2003)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 13–23.
54. Tichelaar S, Ducasse S, Demeyer S. FAMIX and XMI. *Proceedings WCRE 2000 Workshop on Exchange Formats*. IEEE Computer Society Press: Los Alamitos CA, 2000.
55. Zou L, Godfrey M. Detecting merging and splitting using origin analysis. *Proceedings 10th Working Conference on Reverse Engineering (WCRE '03)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 146–154.
56. Zou L, Godfrey M. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 2005; **31**(2):166–181.
57. Riel A. *Object-Oriented Design Heuristics*. Addison Wesley: Boston MA, 1996; 400 pp.
58. Marinescu R. Measurement and quality in object-oriented design. *Doctoral dissertation*. Department of Computer Science, Politehnica University of Timişoara, 2002; 155 pp.
59. Marinescu R. Detection strategies: Metrics-based rules for detecting design flaws. *20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 350–359.
60. Lanza M, Ducasse S. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 2003; **29**(9):782–795.
61. Lanza M, Ducasse S. Codecrawler–an extensible and language independent 2d and 3d software visualization tool. *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series. Franco Angeli: Milano, 2005; 74–94.
62. Arévalo G. High level views in object-oriented systems using formal concept analysis. *Doctoral dissertation*. University of Berne: Berne, 2005; 113 pp.

AUTHORS' BIOGRAPHIES

Tudor Gîrba attained the PhD degree in 2005 at the University of Berne, Switzerland, and since then he is working as senior researcher at the Software Composition Group, University of Berne, Switzerland. His interests lie in the area of software evolution, meta-modeling, reverse engineering, reengineering, information visualization, and quality assurance. He is one of the main developers of Moose, he developed the Van software evolution analysis tool, and participated in the development of several other reverse engineering tools. He also offers consulting services in the area of reengineering and quality assurance.

Stéphane Ducasse is Professor at the Université de Savoie, France, where he leads the Language and Software Evolution group of the LISTIC laboratory. His fields of interests are: design of reflective systems and object-oriented languages, web development and reengineering and evolution of object-oriented applications. He is one of the main developers of the Moose reengineering environment. He is the president of the European Smalltalk User Group and has lot of fun programming in Smalltalk. He wrote several books in French and English: La programmation: une approche fonctionnelle et recursive en Scheme (Eyrolles 1996), Squeak (Eyrolles 2001), Object-Oriented Reengineering Patterns (MKP 2003), Squeak: Learning Programming with Robots (APress 2005), Object-Oriented Metrics in Practice (Springer 2006).