Modeling Software Evolution by Treating History as a First Class Entity

⋆

Stéphane Ducasse [1,4] Tudor Gîrba [2,4]

Software Composition Group
University of Bern, Switzerland

Jean-Marie Favre [3]

LSR-IMAG Laboratory
University of Grenoble, France

Abstract

The histories of software systems hold useful information when reasoning about the systems at hand or about general laws of software evolution. Yet, the approaches developed so far do not rely on an explicit meta-model and do not facilitate the comparison of different evolutions. We argue for the need to define history as a first class entity and propose a meta-model centered around the notion of history. We show the usefulness of our a meta-model by discussing the different analysis it enables.

Key words: software evolution, history meta-model

1 Introduction

The importance of observing and modeling software evolution started to be recognized in 1970's with the work of Lehman[15]. Since then more and more

⋆ In Proceedings of the Workshop on Software Evolution Through Transformations (SETra 2004)

[1] Email: ducasse@iam.unibe.ch
[2] Email: girba@iam.unibe.ch
[3] Email: jean-marie.favre@imag.fr

[4] Ducasse and Gîrba gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004) and "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

research has been spent to identifying the driving forces of software evolution, and to using this information to better understand software. However, the approaches developed so far, do not rely on an explicit meta model for evolution analysis and do not facilitate the comparison of different evolutions.

The goal of this work is to propose a meta-model which allows for the usage of historical information just like any other kind of information.

Before going into details, we define three terms: *version*, *evolution* and *history*. A *version* is a snapshot of an entity at a particular moment in time. The *evolution* is the process that leads from one version to another. A *history* as the reification which encapsulates knowledge about evolution and version information. According to these definitions, we say that we use the history to understand the evolution (*i.e.,* history is a model of evolution).

This paper shows Hismo, a meta-model having in its center the notion of history, and argues that we need such a meta-model to reason about evolution of software systems. As a validation for our approach we present examples of historical measurements and history manipulations and show different usages for reverse engineering.

In the next section we enumerate the requirements a meta-model should support and we analyze existing techniques to analyze software evolution. In Section 3, we introduce Hismo, our history meta-model. In Section 4 we show examples of history measurements and in Section 5 we give examples of analyses enabled by our meta-model. In the end, we draw the conclusions and present the future work.

2 Software Evolution Analyses

Based on our analysis of the field, the requirements that a meta-model for analyzing the evolution of software systems fulfill are:s

- The meta-model should offer means to easily quantify and compare different property evolutions of different entities. For example, we must be able to compare the evolution of number of methods in different classes.
- The meta-model should allow for an analysis to be based on the evolution of different properties. Just like we can now 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, it should be refactored. In the same line, adding or removing the lines of code in a class while preserving the methods might be a sign of a bug-fix.
- The meta-model should provide change information at different level of abstraction such as packages, classes, methods (*i.e.,* not just text modifications).
- The meta-model should provide for the comparison in detail of two distinct versions of the same entity.

- The analysis should be applicable on any group of versions (*i.e.,* we should be able to select any portion of the history).

In the followings we enumerate different techniques used to analyze software evolution and how these techniques relate to the above requirements.

2.1 Evolution Chart Visualization

Since 1970 research is spent on building a theory of evolution by formulating laws based on empirical observations [15] [14]. The observations are based on interpreting evolution charts which represent some property on the vertical (*i.e.,* number of modules) and time on the horizontal (see Figure 1). Lately, the same approach has been employed to understand the evolution of open-source projects [2] [3]

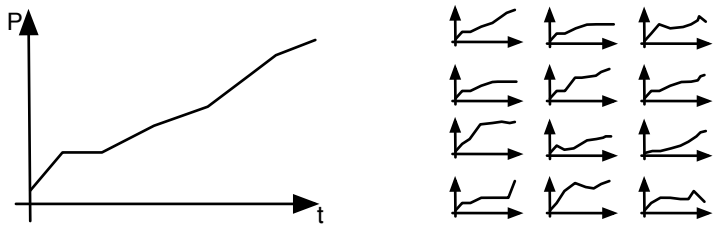

Fig. 1. Evolution chart example with some property on the vertical and time on the horizontal.

This approach is useful when we need to reason in terms of one property, but it makes it difficult to reason in terms of more properties at the same time, and provides only limited ways to compare evolutions of different properties. For example, it is suitable to use this technique to analyze the evolution number of modules in a system, but it is difficult to correlate the number of modules, with the total lines of code and with the number of developers.

In the left part of Figure 1 we display a graph with the evolution of a property P of an entity. From the figure we can draw the conclusion that P is growing in time. In the right part of the figure we displayed the evolution of 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.

2.2 Evolution Matrix Visualization

Visualization has been also used to reason about multiple evolution properties and to compare different evolutions of different entities. Lanza and Ducasse arranged the classes of the history of a system in an Evolution Matrix like in Figure 2 [12]. Each rectangle represents a version of a class and each line holds

all the versions of that 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: pulsar, idle, supernova or white dwarf.

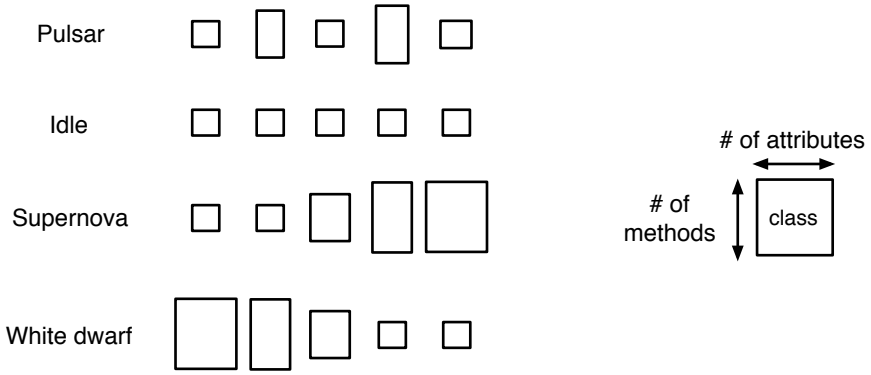

Fig. 2. Samples of class evolution patterns detectable in the Evolution Matrix.

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 (*i.e.,* 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. For example, we would like to know if the pulsar or idle classes are big or not.

Based on the detected patterns we can build a vocabulary for characterizing classes. Thus, in a system we can have pulsar classes or idle classes. But, pulsar and idle characterize a complete line and not just a cell in the matrix. Therefore, pulsar and idle characterize the way a class evolved over time and not a class. Based on this observation we concluded that we need a noun to which to assign the pulsar-like properties: the history.

Other visualizations approaches are based on similar meta-models. Jazayeri analyzes the stability of the architecture [11] by using colors to depict the changes. Taylor and Munro [18] visualizes version data with a technique called *revision towers*. Ball and Eick [1] develope visualizations for showing changes that appear in the source code. Collberg *et al.* use graph-based visualizations to display the changes authors make to class hierarchies [4]. Rysselberghe and Demeyer use a simple visualization based on information in version control systems to provide an overview of the evolution of systems [19].

2.3 Release History Meta-Model

Fischer *et al.* modeled bug reports in relation with version control system (CVS) items [7]. In Figure 3 we present an excerpt of the Release History Meta-model. The purpose of this meta-model is to provide a link between the versioning system to the bug reports.

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. The authors used this meta-model to recover features based on the bug reports [6]. These features get associated with a CVSItem.

A similar meta-model have been used to detect logical coupling between parts of the system [8]. The authors used the CVSItemLogs to detect the parts of the system which change together and then they used this information to define a coupling measurement.

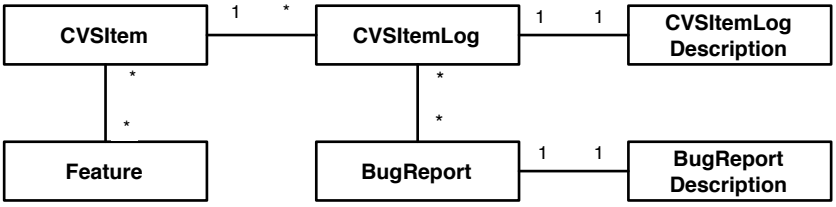

Fig. 3. Excerpt from the Release History Model-model.

The main drawback of this meta-model is that it does not take into consideration the structure of the software at the version level – *i.e.,* the system is represented with only files and folders, but no semantical units are represented (*e.g.,* classes or methods). Therefore, this meta-model does not offer support for different semantics of change – *i.e.,* it gives no information about what exactly changed in a system.

Zimmerman *et al.* aimed to provide mechanism to warn developers that: "Programmers who changed these functions also changed ...". The authors placed their analysis at the level of entities in the meta-model (*e.g.,* methods) [20]. Unfortunately, they did not explicitly describe their underlying meta-model.

3 Hismo - History Meta-Model

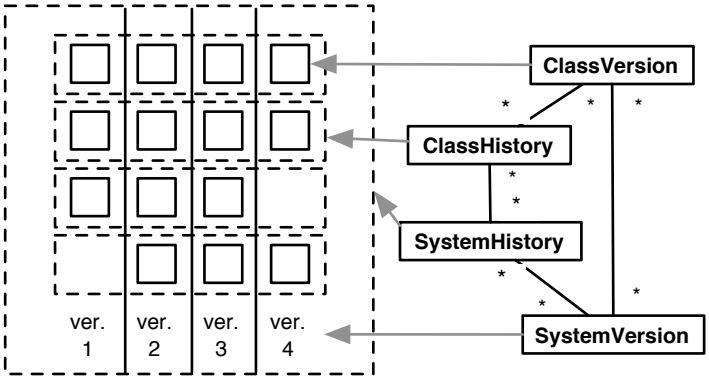

Fig. 4. History and the Evolution Matrix.

Figure 4 shows how a meta-model centered around the notion of history

can be built: each cell in the matrix is a Class Version which makes for each line to represent a Class History. Moreover, the whole matrix is actually a line formed by SystemVersions, which means that the whole matrix can be seen as a SystemHistory. In the right side of the figure we built a small meta-model which shows that a SystemHistory has more ClassHistories.

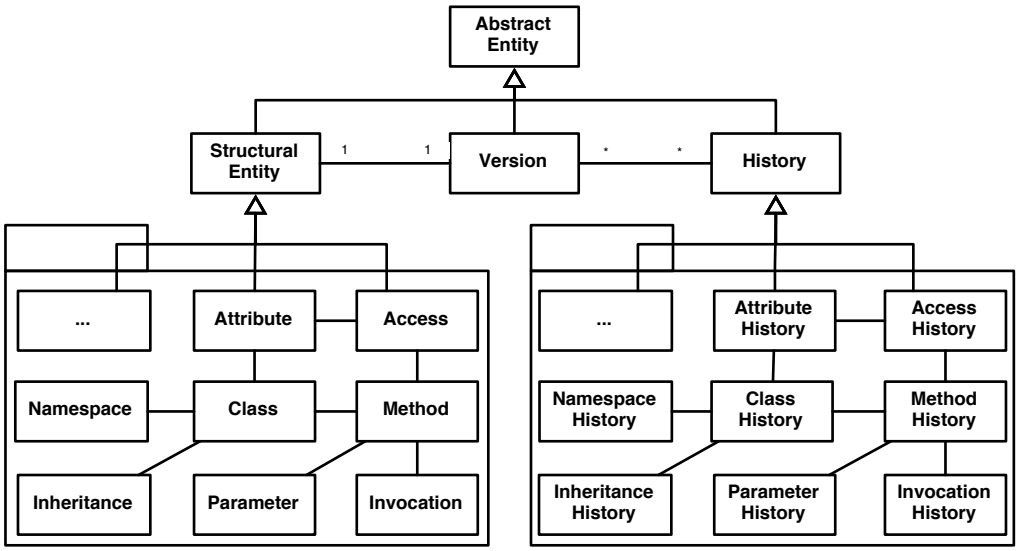

Fig. 5. An excerpt of Hismo and its relation with a source code meta-model. We did not represent all the inheritance relationships to not affect the readability of the picture.

In Figure 5 we show a reduced history meta-model based on a source-code meta-model. In our case we used FAMIX [5]. Each version entity has a correspondent history entity. Also, the relationship at version level (*e.g.,* a Class has more Methods) has a correspondent at the history level (*e.g.,* a ClassHistory has more MethodHistories).

A history does not have direct relation with a version entity, but through a Version wrapper. In Figure 6 we show the details of the relationship between History and Version.

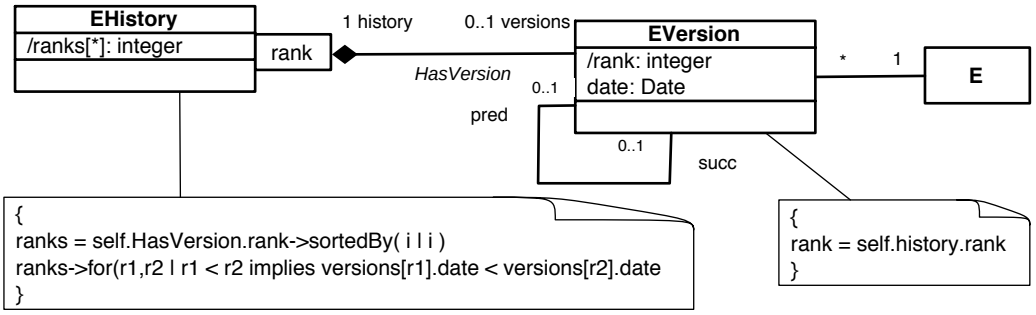

Fig. 6. Details of the relationship between the History, the Version and the structural entity (E). We used OCL notation.

4 History Measurements in Hismo

In this section we show some examples of how we use our meta-model to measure the evolution. We also show how the meta-model supports history selection and how measurements can be applied on any such selection.

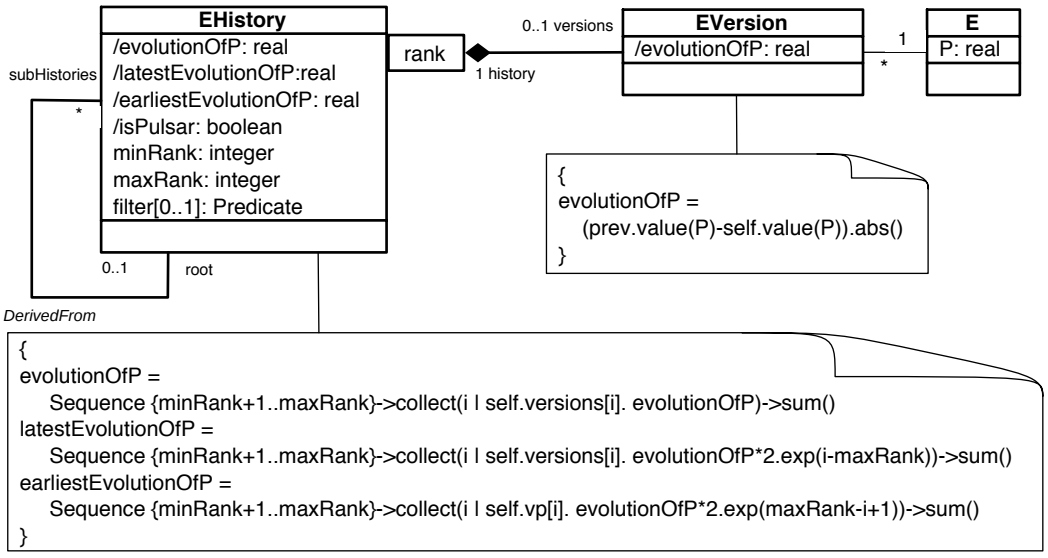

Fig. 7. Examples of history measurements definitions.

In Figure 7 we introduce three measurements: Evolution of P, Latest Evolution of P and Earliest Evolution of P.

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 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.

Figure 7 also shows that given a history we can filter it to obtain a sub history. As the defined measurements are applicable on a history, and a selection of a history is another history, the measurements can be applied on any selection too.

In Figure 8 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

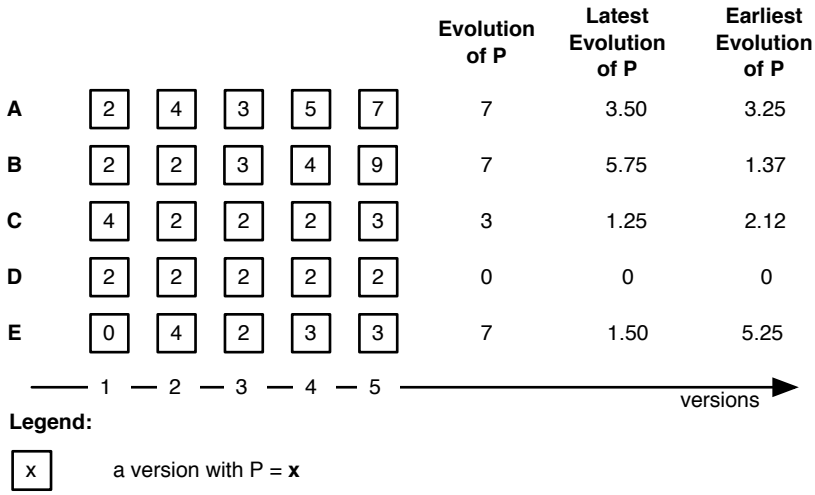

Fig. 8. Example of history measurements.

was changed the same as shown by the Evolution of P (EP). 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 past in the history of class E and (iii) in the history of class A the changes were scattered through the history more evenly.

- 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 P property can be a property like: number of methods of a class, complexity of a method etc. Furthermore, we can define other measurements like: addition/removals of P, stability/instability of P etc.

5 Hismo Applications

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 time into numbers which are assigned to the corresponding histories. In the rest of the section, we describe three applications based on our meta-model:

- Build more complex historical measurements,
- Visualize different historical measurements to determine correlations and patterns of evolution.
- Build automatic queries which combine different evolution characteristics with version information to improve the detection of design flaws.

5.1 Yesterday's Weather

The above mentioned measurements were used to define another measurement: Yesterday's Weather (YW) [10]. YW is defined to be 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 in 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.

YW is a historical measurement obtained by combining different historical measurement which are applied on sub histories.

5.2 Hierarchy Evolution Complexity View

Based on Hismo, a visualization has been proposed to detect patterns of hierarchy evolution [9]. The visualization is based on the polymetric view [13]. Figure 9 shows the visualization applied on the history of six class hierarchies. The nodes represent class histories and the edges inheritance histories. Both the nodes and the edges are annotated with historical measurements. The visualization combines the evolution of different properties for building a vocabulary to characterize the evolution of class hierarchies: old hierarchies, stable hierarchies etc.

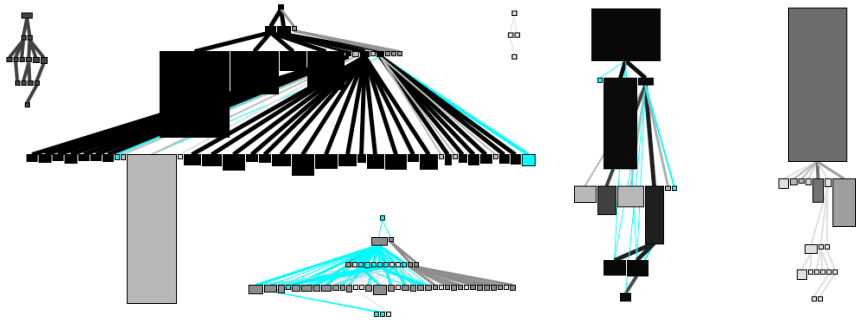

Fig. 9. Examples of class hierarchies evolution. Nodes represent class histories and the edges represent inheritance histories. Node width = Evolution of Number of Methods; Node height = Evolution of Number of Statements; Node color = Class Age; Edge width = Age; Edge color = Age.

5.3 Design Flaws Detection

Another usage of history measurements was proposed for improving design flaws detection [17]. In particular, the work shows how the detection of Data-Classes and GodClasses [16] based on version measurements can be improved by taking into account information like: stability or the persistence of the flaw.

For example, Marinescu defines GodClasses as "those classes that tend to centralize the intelligence of the system." [16]. He also defined measurements-based expressions to detect GodClasses. We used the historical information to qualify GodClasses as being harmless if they were stable for a large part of their history, because that means those classes were not a maintainability problem in the past (*e.g.,* 95%). Below we present the expression we used.

```
context ClassHistory
    derive isHarmlessGodClass: (self.versions->last().isGodClass) &
                               (self.stabilityOfNOM > 0.95)
```

In this expression, we show how we can combine the historical information with other kinds of information to build our reasoning.

6 Conclusions and Future Work

Understanding software evolution is important as evolution holds information that can be used either in reverse engineering or in developing laws of evolution.

We browsed various techniques that have been used to understand the evolution, we discussed their shortcomings and we gathered requirements for our meta-model:

- Comparison of different evolutions of the same property,
- Combination of different property evolutions,
- History navigation/selection,
- Different semantics of change,
- Detailed version comparison.

Based on these requirements we proposed Hismo, a meta-model centered around the notion of history, and we gave examples of measurements applied on history. As a validation we showed the usages of our meta-model in different analyses.

In Figure 10 we show how a GeneralizedHistory is not just a sequence, but a graph; thus we can model branches. In the future, we would also like to explore the information given by branches.

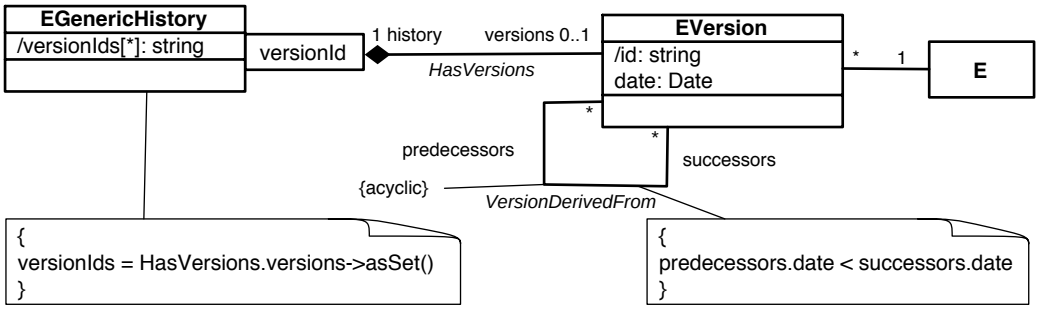

Fig. 10. Generalized Hismo.

References

[1] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, pages 33–43, 1996.

[2] Andrea Capiluppi. Models for the evolution of os projects. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 65–74, 2003.

[3] Andrea Capiluppi, P. Lago, and M. Morisio. Evolution of understandability in oss projects. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 58–66, 2004.

[4] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 77–86. ACM Press, 2003.

[5] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[6] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 90–99, November 2003.

[7] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, September 2003.

[8] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance 1998 (ICSM '98)*, pages 190–198, 1998.

[9] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of ICSM 2004 (International Conference on Software Maintenance)*, 2004.

[10] Tudor Gîrba and Michele Lanza. Visualizing and characterizing the evolution of class hierarchies. In *Fifth International Workshop on Object-Oriented Reengineering (WOOR 2004)*, 2004.

[11] Mehdi Jazayeri. On architectural stability and evolution. In *Reliable Software Technlogies-Ada-Europe 2002*, pages 13–23. Springer Verlag, 2002.

[12] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002 (Langages et Modèles à Objets*, pages 135–149, 2002.

[13] Michele Lanza and Stéphane Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.

[14] Manny M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[15] Manny M. Lehman and Les Belady. *Program Evolution — Processes of Software Change*. London Academic Press, 1985.

[16] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. Ph.D. thesis, Department of Computer Science, "Politehnica" University of Timişoara, 2002.

[17] Daniel Raţiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Proceedings of CSMR 2004 (European Conference on Software Maintenance and Reengineering)*, pages 223–232, 2004.

[18] Christopher M. B. Taylor and Malcolm Munro. Revision towers. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50. IEEE Computer Society, 2002.

[19] Filip Van Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004. to appear.

[20] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, 2004.