Unique Identification of Elements in Evolving Models:

Towards Fine-Grained Traceability in Model-Driven Engineering

Vom Fachbereich Elektrotechnik und Informatik der Universität Siegen

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von

Dipl.-Inform. Sven Wenzel

Siegen, November 2010

Gedruckt auf alterungsbeständigem holz- und säurefreiem Papier.

Als Dissertation genehmigt vom
Fachbereich Elektrotechnik und Informatik
Universität Siegen

Einreichung	3. November 2010
Mündl. Prüfung	4. März 2011
Dekan	Prof. Dr. M. Pacas, Universität Siegen
1. Gutachter	Prof. Dr. U. Kelter, Universität Siegen
2. Gutachter	Prof. T. Systä, PhD, Techn. Universität Tampere, Finnland
3. Gutachter	Prof. Dr. J. Ebert, Universität Koblenz
Vorsitzender	Prof. Dr. V. Blanz, Universität Siegen

Abstract

Model-driven engineering (MDE) is a widely accepted methodology in software engineering. At the same time, the ability to retrace the engineering process is an important success factor for software projects. In MDE, however, such traceability is often impeded by the inadequate management of model evolution. Although models have a very fine-grained structure, their different revisions and variants are prevalently managed as monoliths in a file-based software configuration management (SCM). This causes the identification problem: if the fine-grained elements are not assigned with globally unique identifiers, we cannot identify them over time. If such identifiers would be given, they can be misleading. As a consequence, we cannot comprehend the evolution of elements and traceability relationships among the elements cannot be managed sufficiently.

This thesis presents a novel solution to the identification problem. It establishes a representation to describe the history of a model and its fine-grained elements inside. The key feature of the representation is a new kind of traceability relationship, called *identification links*. They allow us to identify elements of a given revision in other revisions or variants of the model. The identification is even applicable to anonymous elements and model fragments. It provides us with a broad spectrum of opportunities: e.g. management of fine-grained traceability links, evolution analysis, merging of development branches. Due to the expression of model evolution in the history representation, we are further able to capture the changes that have been applied to the traced elements. This thesis further presents an algorithm to infer the identification links automatically. The approach does not rely on persistent identifiers, but it utilizes a similarity-based model comparison technique to locate the model elements in other revisions.

The algorithm and the history representation have been implemented in a prototype. It is metamodel and tool independent and can work with an arbitrary SCM. Existing modeling environments do not have to be modified. Traceability information and evolution information is accessible through a service interface and can thus be integrated in arbitrary tools. The evaluation of our approach by means of controlled experiments with data from real models attested excellent precision and recall values for the identification of model elements over time. Furthermore, different evolution analysis tools have already been built on our approach, which documents the practical applicability of our solution.

Kurzfassung

Die modellgetriebene Entwicklung ist eine weit verbreitete Methode zur Softwareentwicklung. Die ungeeignete Versionierung von Modellen verhindert jedoch oftmals die Nachverfolgbarkeit des Entwicklungsprozesses. Trotz ihrer feinkörnigen Struktur werden Modelle oft monolithisch auf Basis dateibasierter Konfigurationsmanagementsysteme verwaltet. Modellelemente können in diesem Fall nicht über die Zeit hinweg identifiziert werden, weil globale Identifizierer entweder fehlen oder irreführend sein können. Aufgrund dieses Identifikationsproblems können Änderungen an Modellelementen nur sehr schwer nachvollzogen werden. Außerdem können Referenzen zur Nachverfolgbarkeit zwischen Elementen verschiedener Revisionen nicht sinnvoll verwaltet werden.

Diese Dissertation löst das Identifikationsproblem. Sie führt eine Repräsentation ein, mit der die Historie von Modellen und deren feinkörnigen Elementen abgebildet werden kann. Ein zentraler Bestandteil dieser Repräsentation sind *Identifizierungslinks*, die es ermöglichen ein gegebenes Element in anderen Revisionen oder Varianten des Modells wiederzufinden. Der Ansatz unterstützt auch anonyme Elemente und komplette Modellfragmente. Diese neuartige Identifizierung ermöglicht z.B. die Verwaltung von feinkörniger Nachverfolgbarkeitsinformation, die Analyse von Modellevolution oder das Mischen von Entwicklungszweigen. Da die Repräsentation auch die Evolution eines Modells abbilden kann, können die Veränderungen identifizierter Elemente besser erfasst werden. Zudem wird in dieser Dissertation ein Algorithmus entwickelt, mit dem Identifizierungslinks zwischen Modellelementen verschiedener Revisionen inferiert werden. Dieser stützt sich nicht auf persistente Identifizierer, sondern nutzt einen ähnlichkeitsbasierten Differenzalgorithmus, um Elemente in anderen Revisionen wiederzufinden.

Der Algorithmus und die Historienrepräsentation wurden in einem modelltyp- und werkzeugunabhängigen Prototyp implementiert, der mit beliebigen Konfigurationsmanagementsystemen zusammenarbeitet, ohne dass diese angepasst werden müssen. Die Informationen zur Nachverfolgbarkeit und Evolution von Modellelementen sind über eine Programmierschnittstelle abfragbar, die sich in beliebigen Werkzeugen nutzen lässt. Der beschriebene Ansatz wurde mit kontrollierten Experimenten auf Basis realer Modellhistorien erfolgreich evaluiert. Darüber hinaus wurde seine praktische Anwendbarkeit durch verschiedene darauf aufbauende Werkzeuge zur Evolutionsanalyse belegt.

Acknowledgements

I would like to thank my supervisor Professor Udo Kelter for his scientific support and the endless discussions we had. I am also very grateful to my co-supervisors Professor Tarja Systä and Professor Jürgen Ebert who encouraged and supported me during the whole process of writing. They gave a lot of useful feedback and inspiration that improved my thesis.

In addition, I would like to thank my (former) colleagues, Timo Kehrer, Maik Schmidt, Pit Pietsch, Stefan Berlik, Christoph Treude, Jörg Niere, Christian Köhler, Hamed Shariat Yazdi, Dhiah El Dhien Abou-Tair, and Maryam Nasiri, for the discussions, for listening to my ideas, for co-authoring papers, and for proof reading. Special thanks go to Roswitha Eifler and Frank Schuh for their organizational and technical support. They all provided me with a pleasant working atmosphere including memorable coffee breaks with tons of cookies.

For the technical realization, I would like to thank all contributors of the SiDiff project as well as the students Dennis Koch, Hermann Hutter, and Jens Falk for their endurance while listening to my nebulous ideas and implementing them in different prototypes. Furthermore, I would like to thank all students who have supported me directly or indirectly with their diploma theses, their project groups, or their work as student assistants.

I would further like to thank all of my friends and relatives for reminding me of the other important things in life. I am especially grateful to my mother (who died much too early), my father, and my brother for always supporting me and inspiring me to bring out the best in myself.

Last but not least, I wish to express my deepest gratitute to my beloved wife for her endless love, her patient care, and her unconditional support, and to my dear children for filling me with joy and happiness. Thank you!

There is no branch of detective science
which is so important and so much neglected
as the art of tracing footsteps.

— Sherlock Holmes
in "A Study in Scarlet" by Sir Arthur Conan Doyle

Contents

Part I

Overview

Chapter 1

Introduction

The IEEE Standard Glossary of Software Engineering Terminology defines software engineering as

> the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. [63]

Modeling is a well-established methodology in the field of engineering. In software engineering, modeling is used, too. In model-driven engineering the processes are characterized by the ubiquitous application of modeling. Depending on the application domain, engineering disciplines often require the ability to retrace the complete engineering process; e.g. the ISO standard 26262 [65] regulates the traceability between requirements and tests in automotive engineering. However, traceability is still an unsolved problem if it concerns the fine-grained elements contained in the models.

1.1 Model-Driven Engineering

Models have an inherent position in software engineering. A model is the abstract representation of a more complex original. It can be used in a descriptive way to mirror an original, or in a prescriptive way to specify something to be created [85]. Descriptive models are mainly used to explain a problem, to document it, and to build a basis for communication and analysis. Prescriptive models are not limited to specification, but they even support the generation of software systems. The generative usage of models leads to model-driven (software) engineering[1] (MDE).

Recently, model-driven engineering has become a widely accepted methodology in software development. It is especially applied in safety-critical domains such as

[1] It is also called model-driven development. In contrast, the term *model-based* often describes a software development, in which models are intensively used for different tasks except for generation, e.g. documentation or testing. However, there is not a clear differentiation between both terms.

automotive and aerospace. Developers work mainly or only with models. Models are no longer only the documentation or specification of the system. They *are* the system. Models have been proved to provide good support for precise definition of the planned software system, which even enables an automation of the software development process. Initial specification models describe the requirements, and they prescribe subsequent models and other documents of the development process. They are refined step-wise towards certain domains and platforms. Finally, they are transformed into executable code or they can even be interpreted directly. Developers can thus work on a level which is independent from any target platform. The adaptation onto specific environments can be done by generators. Reuse of models is achieved.

An example is the model-driven architecture (MDA) initiative of the OMG [104]. It defines a high degree of automation throughout the software development process. Based on a computation independent model that describes the requirements of the software to be developed, the developers can create platform-independent models (PIM) that describe the functional aspects of the software without describing how the underlying platform is used. Transformations are used to translate the PIM into platform-specific models (PSM) that describe the functioning on different platforms [110]. The term platform has to be seen in an abstract way as it does not necessarily relate to an execution environment. A PSM can rather be a PIM for other transformations. MDA always presumes an object-oriented view on the software. However, in this thesis we focus on model-driven engineering in general, which has a wider scope than MDA. It focuses on the complete engineering process including different paradigms, languages, and tools [71].

1.2 Model Evolution

Engineering of software is a long and complex task. Even in MDE, software passes various stages of development. Similarly to software evolution, which is widely understood as the change of software over time [16, 80], we can define *model evolution* as the change of models over time. Software evolution often only refers to the stage in the system's life cycle that takes place after initial development [14]. In MDE, however, the first model *is* already the software system, even if it is very abstract. Hence, the term model evolution includes initial development, too.

We can distinguish two kinds of model evolution. On the one hand we have model transformations that transform models to a more specific level of abstraction or enrich them with additional data. Transformations describe well-defined changes that are applied to models and they allow us to automate many parts of the development process. On the other hand there is still a significant amount

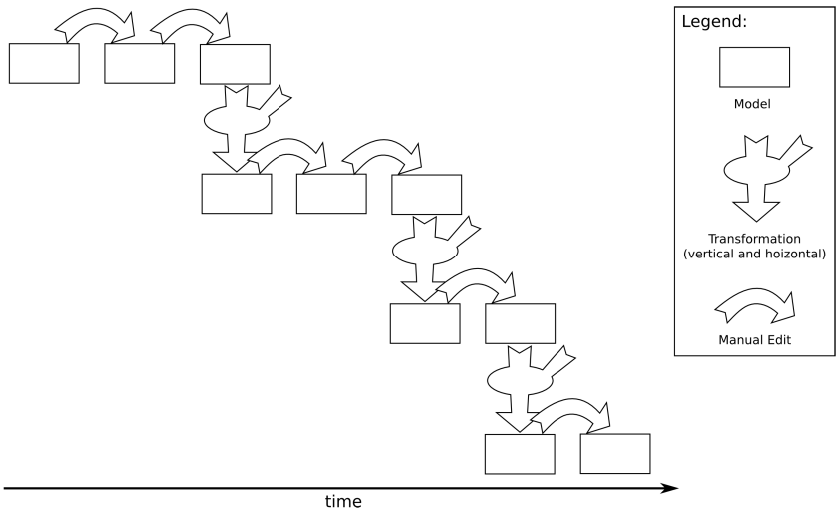


Figure 1.1: *The MDE process with manual editing*

of manual editing applied to models. The initial models are rarely designed in one step, but rather many iterations are required; especially in agile development processes [21]. Furthermore, intermediate results of the different transformations may require manual corrections, e.g. if a transformation engine does not support a particular target platform. As a consequence, models evolve continuously.

Figure 1.1 illustrates an MDE process that includes manual editing. A model can thus be changed in two directions. In vertical direction we see evolution that originates from model transformations. The horizontal direction indicates manually applied changes. Although the illustration seems to imply that all transformations are vertical, transformations can also be horizontal (i.e. they change a model within one level of abstraction). If we imagine a time line going horizontally from left to right, we can understand manual editing as a process that is applied over time, while transformations are applied instantly. The transformations can be seen as *controlled evolution*. Each transformation is well-defined and its purpose is often described in detail. Often we can even define the inverse that reverses the changes of the original transformation.

In contrast to transformations, manual editing is to some extent arbitrary and undetermined. It leads to *uncontrolled evolution*, since there is usually no detailed description of the changes. In the best case, we have vague, natural language information about the intention of the change, i.e. an associated change request or a commit message in a configuration management system. As a consequence, it is difficult to comprehend the changes applied to a model and to retrace the development of a software system. In the remainder of this thesis we only deal with model evolution that is caused by manual editing.

1.3 Traceability

Traceability is very important in software engineering. It allows us to keep track of relationships among different documents involved in the engineering process. These relationships are utilized for comprehension, estimation of change impact, testing, monitoring progress, reuse, and many other interests [8]. The IEEE Standard Glossary of Software Engineering Terminology defines traceability as:

> The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another. [63]

Traceability is often mentioned in conjunction with requirements. In this case, requirements of a software are linked backwards to the stakeholder's rationale and forward to software artifacts implementing them [54]. This provides information that can be used to understand why the software has been implemented in a specific way or how a feature has to be tested.

However, traceability is not limited to requirements. It rather covers many different types of relationships (also called *traceability links*) that can be established between documents or between software elements inside these documents. Another example is logical coupling. It provides information about entities that are often changed together [156]. Further examples of traceability links are the relationships between design decisions and the rationale behind them [131, 150], and dependencies between software artifacts [38].

Some types of traceability links, such as logical coupling, can be inferred from software documents directly. Most relationships can only be established with external knowledge, e.g. if requirements are traced through the complete development process or if the rationale behind design decisions is captured. Capturing this knowledge is often a long-lasting and expensive task. Hence, the traceability links need to be managed in a sufficient way, e.g. in separate documents or dedicated tools/databases. The links have then references that point to the linked documents or entities by using names, path expressions, or other identifiers.

Traceability is often given implicitly by names or abbreviations that are used in different documents. It is assumed that equal names refer to the same entity inside given namespaces. This assumption is often made unconsciously and most people use this kind of traceability thoroughly, although they rarely entitle it. This concept heavily relies on the conventions of a software project and can easily lead to ambiguities, misunderstandings, and even worse problems.

A wide overview on the different types of traceability, their application, and approaches to capture and manage traceability links is given in the surveys of Spanoudakis and Zisman [134], von Knethen and Paech [143], and Winkler and von Pilgrim [149].

1.4 Traceability in MDE: A Challenge

A lot of research has been invested in traceability in model-driven engineering [1, 2]. Most traceability approaches known from traditional, code-driven development can easily be adopted to MDE. The relationships between model documents do not differ from relationships between other artifacts of the development process. Their management is well handled, since model documents have already been part of the early phases of software development processes for a long time.

Traceability links between model elements, however, have to be managed differently. In some cases, links between elements of one model can be expressed by the modeling language directly, e.g. dependencies, which are specific model elements in the UML. In other cases, such as relationships between elements of different models, the links have to be stored separately. We can differentiate between internal and external link storage. Since internal storage requires the extension of metamodels with new link elements and original models get enlarged by mixing user data with traceability information, traceability links are prevalently stored externally in separate models or designated tools/databases. The information can be merged into the models on demand [73].

With respect to model evolution caused by manual editing, however, the enforcement of traceability support in MDE becomes a challenge. External storage and links between different models require the ability to uniquely identify entities inside the models [27]. Thereby, we get into a quandary. Tracing entities over time is basically very similar to tracing them across different documents. It is just a special case where the different documents have a predecessor-successor relationship, i.e. they are different revisions of the same document. Hence, we need traceability in order to identify entities at different times. But on the other side, we need to be able to identify entities at different times in order to support traceability in general. The tracing of entities inside evolving documents is thus a very crucial factor if we want to be able to retrace the complete engineering process. This kind of traceability is often equated with *identification*.

Definition 1.1: Given a model element e, **identification** is the process of locating this element in other revisions of the model; i.e. recognizing model elements in other revisions of that model to be e at different times.

If model evolution is caused by transformations, the transformations can be used to generate appropriate information for identification. In case of manual editing, information for identification is missing. Existing traceability approaches for textual documents cannot be adapted to evolving models. The particularities of models compared to source code are in conflict with the assumption of existing traceability approaches, that we can directly identify model elements at different times. The identification is hampered by different factors that have their origin in the differences between models and textual documents. We discuss these problems in what follows.

1.4.1 Lower Significance of Identifiers

As said before, traceability links are often based on the identifiers of software entities, i.e. they are used to point to the linked entities. In models, however, identifiers have a lower significance than in source code. Source code strongly uses names to express references between software elements, e.g. a statement refers to the methods it calls by showing their names, and parameters are given by names of variables which store the parameters. Hence, names play an important role in source code. A large amount of names is even required to be unique in particular contexts, otherwise the identification of elements would not work.

Models can express references and relationships by designated elements. An example is an association in UML [108]: it connects two classes in order to express that one class knows the other. Names play a subordinate role and they are not always required to be unique. The “DataElement” objects in ASCET [40], for example, have names only for documentation purposes; the values are meaningless from a technical point of view. In many cases model elements can even be anonymous, i.e. they do not have any attribute for identification; they are just identifiable by their neighborhood. Identifiers are only necessary if model elements refer to elements of other models e.g. if a large (maybe virtual) model is separated into multiple small models that are stored in different files. Then the identifiers refer to elements of other models.

While we can assume that identifiers within source code are rarely changed [7] and that they can thus be used to address entities in different revisions, the identifiers of model elements often change arbitrarily from one revision of the model to another, so that they are not sufficient for addressing the elements over time.

1.4.2 Representation of Models

If identifiers are not present, one could think about identification based on pointers into file representations, e.g. “the element represented by line 42 in the file

'model.xmi"'. This approach might be sufficient for source code, but it is not for models. Although source code is only a low-level representation of a software, this representation is fairly consistent with syntax trees, because large parts of programs are sequences of statements. Modifications of the programs map reasonably well (although not perfectly) onto changes in the textual representations. The changes are mostly local, so that other parts of the source code are not affected. In most cases, the files keep their structure, and entities remain at the same relative position to other elements.

Things are quite different in the case of models, which exist in many different, mostly graphical languages. Independent of any particular language, models can conceptually be seen as graphs. In most file representations of graphs, such as XML serializations[2], we rarely find structures with a linear order similar to sequences of statements in code. There are rather elements (vertices) which are connected by different relations (edges). The mapping of models onto textual file representations is arbitrary and differs for each particular modeling language. Especially with respect to evolution, the file representation of models can lead to problems. Single modifications applied to models often result in many changes at widespread positions within the file representation. Hence, the evolution of a model cannot be mapped onto the file representation exactly. Furthermore, the semantics of a model element are often determined by the context of the element, i.e. its neighborhood. If we look at the plain file representation without interpreting of the data, however, elements that are neighbored in the model are not necessarily neighbored in the file representation. A model element might have evolved although the particular section in the file representation has not changed at all. Hence, we cannot enable traceability by pointers into files, because the position in files is arbitrary and can change from one revision to another. Furthermore, due to the broad spectrum of modeling languages, we would have to implement a new approach for each file format.

1.4.3 Management of Model Evolution

In software engineering, evolution is mainly managed with software configuration management systems (SCM) [9]. A key feature of SCMs is the management of successor-relationships between system versions and archiving of different revisions of files. While most SCMs, e.g. [44, 82, 119, 137], focus on management of textual documents such as source code or binary files, special configuration management systems for models exist, too (see Section 2.1.2). In daily practice, however, these systems hardly find acceptance. In most development departments

[2]Obviously, models can also be stored in binary formats. However, we expect any readable format.

traditional SCMs are already existent due to code-driven development, which is still the dominant portion of software development. Even if code-driven development is replaced by model-driven development, this is not done instantly. Hence, configuration management for both, source code and models, is needed in parallel.

Model configuration management systems are specialized for models and do not support source code. However, most traditional SCMs support the management of arbitrary files. As a consequence, models are prevalently managed with traditional SCMs. Therefore, the models are exported from modeling tools into XMI files [106], or proprietary serialization formats are used to represent the models as files. These files are then put under version control of a SCM. It should be mentioned that the SCM has no information about the content of a file it manages. Thus, the models are seen as monoliths; information about elements or artifacts inside the models is not present. As a consequence, the evolution of the content of models is not visible. We can hardly comprehend what has changed in a model from one revision to another, as we only recognize a change of the model as a whole. Concerning traceability, we cannot even say whether a particular model element is still existent in the newer revision of a model.

1.5 Thesis Objective: The Identification Problem

The previously discussed differences between models and source code and the fact that models are managed in file-based SCMs impedes the identification of model elements (cf. Definition 1.1).

> **Definition 1.2:** The problem that we cannot trustworthily identify model elements across multiple revisions of a model is called the **identification problem**.

Since names play a subordinate role and are not necessarily unique, they are not sufficiently eligible for identification, and they obviously fail at the identification of anonymous elements. Identification with positions inside the file representation similar to line numbers in source code does not work either. The file representation of a graph is arbitrary with respect to its structure. Positions of elements inside diagrams are also useless, because they rarely have a semantic meaning and might arbitrarily change from one revision to another.

Aizenbud-Reshef *et al.* discussed the state-of-the-art of traceability in model-driven development [3]. In their discussion, they define identification as a major problem: *"artifacts may not always have a unique identifier, especially if their granularity is smaller than physically stored artifacts"*. They furthermore state that *"the most challenging aspects of traceability is how to maintain the [...] relationships*

while the artifacts continue to change and evolve". They claim a solution to the identification problem in order to make traceability applicable in MDE: *[An] "issue that needs to be addressed is the ability to uniquely identify artifacts across space and time. Tools typically do not do this, but future tools will need to"* [3].

1.5.1 Typical Scenarios

The identification problem arises in different scenarios. We briefly sketch some of these in what follows.

External Links. A typical scenario is that some model elements are linked by external tools, e.g. elements are linked with a requirement stored in some requirements engineering tool. After the links have initially been set, the model is continuously changed, variants are built, and mostly different developers with different tools work on that model using some configuration management system to share the model. Then, at some later point of time, someone wants to access the elements, which have earlier been linked with the requirement, in the current revision of the model. Due to the fact that the model has evolved and many elements including those implementing the requirement have been changed, it is difficult to trustworthily identify the elements that have been assigned by the external tool before.

Obviously, this case is not limited to requirements, but it arises with all kinds of external knowledge and with links between different models, e.g. originating from model integration or weaving.

Evolution Analysis. Analyzing the evolution of a software system is an important task in software maintenance and in re-engineering [16, 61, 62, 64]. If the system has been developed model-driven, this requires an analysis of the models. It is necessary to identify each single element within the whole history of the models of the system, i.e. to trace the elements from their creation to the current model revision or to the model revision in which they have been removed. If such an identification is given, we can for instance compute metrics for the same element at different times.

Metamodel Evolution. Metamodels evolve, too [43]. The UML, for instance, exists since more than a decade and reached version 2.3 already. Although different versions of a metamodel have a predecessor-successor relationship, the metamodels differ significantly. They lead to new model types and require new modeling tools. The activity diagrams of UML versions 1.x and 2.x, for example, share basically the names of some elements, but the semantics have changed. In other cases, the different versions of a metamodel can be compatible. The UML class models, for example, have basically only been extended in the newer versions. However,

even the extension requires new modeling tools which in turn lead to a break in the evolution of models. For a thorough evolution analysis this gap needs to be closed, too.

Model Merging. It is common practice that software is developed collaboratively in distributed teams. One of the key problems of collaborative work on shared documents is the merging of parallel changes or different development branches [13, 92, 126]. In order to merge variants sufficiently, it is essential to identify the elements that exist in the variants *and* the base revision.[3] If many revisions exist between the base revision and the variants to be merged, it is in addition necessary to check if the elements are really the same. Furthermore, if models are managed in file-based configuration management systems, which are not model-aware, traceability is required to map trustworthily between the elements of the base revision and the elements of the variants.

Bug Tracing. Errors occur in model-driven engineering, too. Hence, if an error (e.g. elements affected by a bug) has been found, it is important to discover its origin and the reason of the occurrence. Therefore, one needs to know the model revision in which the error existed the first time. If the model exists in variants, it is also necessary to know if the other variants (e.g. in software product lines) contain the error, too. Only if all occurrences of the error can be identified, the error can be fixed faithfully.

1.5.2 Traceability-Related Questions in Daily Practice

Obviously, the identification of model elements is an important issue for the design and the implementation of modeling environments and tools that are meant to be used in model-driven engineering. The identification problem also affects many (even simple) questions of developers working with evolving models. Examples of traceability-related questions in model-driven engineering are:

- How old is the given element? Since when does it exist?
- In which revisions and/or variants of the model does the given element exist?
- How much/often has the given element been changed from a certain point in the past until now?
- Are the given element of the given model revision and the equally named element in another model revision the same?

Obviously, developers are not only interested in the tracing of single elements. Often, groups of elements build a conceptual unit and thus they should be seen

[3] We assume a three-way merging, which is the prevalent case if branches are merged.

together. Hence, in all questions mentioned above we can replace the term *"element"* by *"group of elements"*. Furthermore, we can extend the questions related to existence:

- Since when does the group exists? Did it appear in one step or has it "grown"?
- Has the group disappeared totally or have just some elements "left" the group?
- Have the elements of the group changed their connections among each other?

It should be mentioned that a group is not necessarily a coherent model fragment such as a set of states connected by transitions. It is rather possible that developers want to trace sets of elements that are slices of a model and not explicitly connected, e.g. if they look for the origin of logical coupling.

1.6 Thesis Contributions

This thesis presents a new solution to the identification problem that we discussed above. The solution enables the unique identification of single (possibly anonymous) elements or fragments in evolving models, even if they are managed in file-based software configuration management systems.[4]

- As part of the solution we establish a novel **data model to describe the history of a model**. The description enables the representation of fine-grained model elements and their evolution. It is independent from particular modeling languages.
- We also present **identification links** as a new type of traceability links. They express the ancestor-descendant relationships between single (possibly anonymous) model elements. Identification links are stored inside the history data model and allow us to trace model elements and fragments from one revision of a model to other revisions and variants.
- Another part of our solution is a newly developed **algorithm that automatically computes identification links** between model elements being the same entity at different times, i.e. the elements are corresponding. The algorithm is configurable to all types of models that can conceptually be seen as graphs. It **does not rely on persistent identifiers**, but rather utilizes a similarity-based model comparison technique to reveal the correspondences between elements.

[4] Although the contributions are subsequently described by using 1st person plural, all contributions originate solely from the author of this thesis.

- Since the underlying model comparison technique is based on heuristics, we define a new **reliability measure to capture the quality** of the reported correspondences. It allows us to assess the reliability of identification links.
- We also define **difference metrics** to express model evolution in numbers. The metrics allow us to measure the amount of change that has been applied to model elements over time.

We have implemented our approach as an information system that holds the history of a model. The core component is an OSGi-based service that (a) enables the import of model revisions from different origins, (b) computes identification links and evolution information, and (c) provides an interface to access the data, e.g. to identify a given element in another revision or to query the changes that have been applied to a model fragment over time. We further have developed an interactive Eclipse-based tool as a graphical user interface to access the data.

We have evaluated the computation of identification links in different controlled experiments. The quantitative and qualitative analysis of our links included a comparison to alternative traceability information and revealed precision and recall values of 98%, and 99% respectively, or better. Additional case studies have attested the practical applicability of our solution in different analysis tools.

1.7 Thesis Structure

In the next chapter we introduce the state-of-the-art of the identification of elements in evolving models. We further analyze different traceability approaches whether they can be adapted to solve the identification problem. In Chapter 3 we capture the requirements of a sufficient solution to the identification problem and we show the core principles of our approach by means of an example history.

We introduce model matching in Chapter 4. It builds the basis of our approach. The generic representation of models as graphs is described in Chapter 5.

The core of our approach (i.e. the representation of history information) is presented in Chapter 6. The algorithms for computing the traceability information are described in Chapter 7. Chapter 8 discusses how the reliability of the computed information can be assessed and how the computed data can be manipulated. The computation of evolution information is presented in Chapter 9. Afterwards, in Chapter 10 we show how the computed information is used to identify model elements or artifacts over time and how their evolution can be assessed.

The implementation of our solution is introduced in Chapter 11. Chapter 12 presents the results of the evaluation. Finally, in Chapter 13 we conclude with an overview on our work and we discuss starting points for future work.

Chapter 2

State-of-the-Art

The identification of model elements across multiple revisions of a model is so far an unsolved problem (i.e. the *identification problem*). In practice, two strategies to avoid the problem exist: living with the limitation of persistent identifiers or using dedicated model repositories. We discuss these strategies and show their limitations in Section 2.1.

In fact, a real solution of the identification problem is – to the best of our knowledge – not existent. However, two related approaches in code-driven development exist that deal with the problem of tracing source code entities across evolution. They are closely related to the identification problem in MDE. We discuss them in Section 2.2.

Although no approach exists that provides identification of model elements over time, e.g. by computing traceability links between different revisions of evolving models, it is worth taking a look at other approaches that deal with other kinds of traceability in model-driven engineering and approaches to recover or maintain traceability links in code-driven development. We discuss some of these approaches in Section 2.3 in order to analyze their applicability to the identification problem.

Further vaguely related work is given by approaches to model comparison. However, we do not present a new approach to model comparison, but we only utilize an existing approach. As a consequence, we do not discuss model comparison approaches here. An excursus to existing model comparison approaches is later given in Section 4.2.

2.1 Avoidance of the Identification Problem

The identification problem can be avoided if persistent identifiers are used, or if the models are managed in dedicated model repositories. We subsequently discuss these approaches.

2.1.1 Persistent Identifiers

It is often argued that the problem of identification is just caused by using the wrong tools. It would not arise if we exclusively use tools that preserve persistent identifiers.

In this case, each element within the model is tagged with a unique identifier. The tools that are used to manipulate the models are required to preserve these identifiers. Many modeling tools such as the IBM Rational Software Architect [60] support identifiers that are persistent with the storage of a model. Whenever a model element is created, it is enriched with a new identifier. The identifier remains the same even if the model is loaded, manipulated, and saved in different tool sessions. Different revisions of a model can be managed with file-based versioning systems such as CVS or Subversion. The models are simply managed by putting their serialization files (e.g. in XMI format) under revision control.

Although it seems that we can avoid the identification problem with the usage of persistent identifiers and the respective tools, this strategy brings along some problems.

Heterogeneous tool landscapes are hardly supported, because each tool must be able to use the identifiers stored in the serialization file, and – even more importantly – to preserve it when writing the model back into a file. As a consequence, the models can hardly be edited with different tools. Regarding configuration management systems, at check-in time all elements of a model must have the same identifier as during check-out. Furthermore, the required homogeneity of tools can even contradict the development process. Using the same tools might be acceptable for single departments, but in global software engineering it is unrealistic to assume such a homogeneity. Improvement of development processes is also contradicted, because the exchange of tools may destroy traceability.

In turn, the feature of preserving identifiers can be negative, too. If an element is changed in order to play another role with different semantics within a model, the preservation of the identifier will lead to useless or even incorrect traceability information. An example is a UML class that is totally changed (i.e. its name, attributes, and operations are replaced completely by others): due to its unchanged identifier, it will remain the same class, although from the semantic point of view no developer would consider that to be the same class at all.

In collaborative work concurrent changes of different developers occur very often. That is also the case if the development contains branching and merging e.g. to develop features in a sandbox and to later integrate them into the trunk. Given the case that two developers create the same element, e.g. an association with identical attributes, each of the model elements would be assigned with different

identifiers. Later, e.g. in the course of merging both changes, the identifier-based tools would work with two different elements (i.e. they have different identifiers), although they are in fact the same element; hence, the merged revision would contain duplicates.

A similar problem occurs if an element is (maybe accidentally) deleted and later recreated with exactly the same properties. In this case, the element cannot be traced, because the recreated element got a new, different identifier. Hence, when looking at the model from an identifier-based point of view, one element has been deleted and *another one* has been inserted.

A naive solution would be to generate identifiers from the properties of elements, which is done for instance in MATLAB/Simulink™[136]. Identifiers are actually given by the element's names or by other local properties of the elements. Renaming or changing other properties of an element would result in new identifiers, so that all advantages of persistent identifiers fail. The approach also fails with model elements that do not contain local properties, such as generalization edges or pseudo states in case of the UML.

In summary, we can say that persistent identifiers come with risks. Although some developers argue that the problems mentioned above do not matter in their work, we cannot neglect them. With the application of model-driven engineering in safety-critical systems, e.g. braking-systems for cars, and the increasing amount of distributed development, we have to look for better alternatives.

2.1.2 Model Repositories

Another approach to avoid the identification problem is the use of dedicated model repositories that support distributed development. Recently different identifier-preserving configuration management systems for models have been proposed. Although they share most of the drawbacks of persistent identifiers and have a limited acceptance in industrial practice [13], we briefly want to introduce the state-of-the-art of model repositories.

2.1.2.1 Stand-Alone Repositories

Oliveira *et al.* have introduced a version control system for UML models called Odyssey-VCS [116]. The system serves as a repository for model files. The models can be edited with various CASE tools and saved in XMI. Internally, Odyssey-VCS transforms the XMI files, which are just used for transport, into an object network. The CASE tools used for modification have to preserve the identifiers given in the XMI files and they are expected to use the same XMI serialization schema. If we extend this approach by using filters to transform different types of model

serialization files into the same internal object representation, we can afford heterogeneous development environments as different modeling tools may be used. However, the tools still have to preserve the identifiers used in the transport files.

Murta *et al.* proposed an improved version of Odyssey-VCS in [96]. The ability to preserve identifiers has been enriched by using UML profiles. A new stereotype allows us to store the identifiers beside the model elements. They are lifted from serialization data to user data. This is similar to internal traceability link storage and comes with the drawback of polluting the models. The approach is not applicable to other modeling languages than UML, as stereotyping or other annotation mechanisms are required.

A similar approach has been proposed by De Lucia *et al.*. They have developed a management system for different kinds of software artifacts, called ADAMS [23]. The system has an extension named COMOVER [12, 25] that allows concurrent model versioning by transforming model files, namely XMI serializations, into fine-grained artifacts manageable by the system. Again persistent identifiers are used to identify each single artifact or model element. However, ADAMS is more than a pure versioning system since it supports the definition of dependency links in order to support traceability in context-aware change management; e.g. when changing an artifact with dependents, the user is asked to change or at least check the dependent artifacts, too.

2.1.2.2 Repositories with Tool Integration

The repositories mentioned above are often referred to as being *state-based*, because they only manage the revisions of models (i.e. their different states). In contrast, *operation-based* repositories do not store the revisions, but they store the edit operations that have been applied to models. They are thus more fine-grained, however, they require the integration into modeling tools.

Schneider *et al.* have developed a library for concurrent object replication, called CoObRA [127, 128]. Integrated in the UML modeling tool FUJABA [47] it provides the ability to manage models in a model repository that stores the models as an internal object network. Model files are not exchanged between different tool instances, but the changes that are made within the editor are recorded and stored in an edit script, i.e. an ordered list of the edit operations applied by the user. The edit scripts can be exchanged between different tool instances and the repository, and changes are applied to the internal object representations.

A similar approach is proposed by the Sysiphus project [15, 72]. It comes with a configuration management system that supports the versioning of a unified software model which integrates different UML model types as well as requirements

and rationale [150]. As in the case of CoObRA, the edit operations made within the tool are recorded.

Both the Sysiphus and the CoObRA approach are tailored for specific model types. Adaptation to other model types requires the ability to hook into modeling tools in order to capture the edit operations applied to models. Both approaches require persistent identifiers. They are used as pointers in edit scripts in order to correctly describe the changes applied to a model.

2.1.2.3 Other Repositories

Further approaches providing model repositories can be found in [19, 33, 103, 112, 124]. From conceptual point of view we can summarizingly say that a model repository can be compared to an object-oriented DBMS, in which model elements are stored as first class citizens and the storage address of objects in the OO-DBMS serves as persistent identifier. The drawbacks of persistent identifiers have been discussed before.

2.1.3 Middleware Solutions

The OPHELIA project [132, 57] is not a model repository but aims at the integration of heterogeneous tools in distributed software engineering. It provides a middleware for unified representation of different software artifacts and another layer for managing relations among them, i.e. traceability links. However, the approach requires modeling tools and environments to implement certain interfaces, which cannot be expected from commercial tools. Furthermore, the unified artifact representation is founded on CORBA objects [109], which in turn is similar to the usage of persistent identifiers.

2.2 Related Approaches in Code-Driven Development

Subsequently we want to discuss two related approaches that are resident in code-driven development. The first one is very related to our problem as it deals with the identification of renamed entities in evolving source code. The second one deals with the analysis of structural software evolution in general. Although both approaches do not explicitly focus on traceability, they basically deal with the identification problem in code-driven development.

2.2.1 Origin Analysis

Godfrey and Tu [51] deal with the problem of tracing source code entities over time. The main objective of the so-called *origin analysis* is the structural evolution of a software system and dissociation from identifiers. It deals with renaming or moving of code entities, which often lead to the assumption that one element has been deleted and another one has been created. Entities that exist in one revision r but do not exist in the ancestor revision are compared to those entities that exist in the ancestor revision but not in revision r. A clone detection technique based on software product metrics is used to analyze whether two suspect entities could actually be the same entity.

Although their approach resides in the domain of source code evolution, parts of the concepts can be transferred to model-driven engineering; their focus on structural evolution is related to typical evolution in model-driven development. However, it has yet not been investigated whether that approach can be applied to models. Furthermore, software product metrics might be sufficient for identification of similar or equal code entities, but they are not solely sufficient for identification of model elements. We can very precisely describe a class in object-oriented code by its complexity and the number of other classes referenced or used by its statements. For a UML class, we cannot compute such metrics, and simple metrics that count model elements, e.g. the number of attributes or operations of a class, lead to ambiguity. They can be used only to reduce the number of candidates [138]. Especially for an automated solution, the ambiguity of the metrics-based detection contradicts a trustworthy identification.

In [52], Godfrey and Zou improved the origin analysis as they considered the merge and split of code entities. Thereby their approach becomes more fine-grained and content of entities is considered. For instance, it can be recognized that an operation replaces two other operations by merging their functionality. However, applicability to model-driven engineering is still not given.

2.2.2 Evolution Analysis

Xing and Stroulia presented an approach to analyze the evolution of object-oriented systems [154]. They apply a pairwise comparison of subsequent reverse-engineered design models from repository snapshots of Java software. The result is a set of change trees describing the changes from one revision to another. They implicitly identify single elements of one revision in another, because the change trees contain information about corresponding elements in the different models. However, the approach only deals with reverse-engineered class models. The evolution of source code is translated into models. The models mainly contain the structural

design of the Java systems, which is rather stable, i.e. the number of changes between the models stays at a moderate level. This particular evolution is not comparable to the evolution of models in model-driven engineering. Furthermore, the approach does not make the traceability information explicit, and neither is the information assessed in any way. Hence, the approach does not provide information on the reliability of the identification of single elements. Groups of elements (e.g. model fragments) cannot be traced at all.

The comparison engine that is used for the evolution analysis has also been published as a separate tool. We discuss it together with other model comparison approaches in more detail in Section 4.2.

2.3 Approaches to Other Kinds of Traceability

The identification problem can neither be sufficiently avoided in daily practice nor has it been solved for model-driven engineering. To the best of our knowledge no such approach exists, but many approaches exist that deal with traceability in general. Hence, we introduce some representative approaches and discuss whether they can be adapted to the identification problem.[1]

2.3.1 Traceability Links for Evolution

First approaches towards traceability links for evolution have been presented by Pohl, who introduced evolutionary links [120], and by Ramamoorthy *et al.*, who introduced history links [122]. These links establish relationships between different revisions of a document and allow us to identify the same document at different times. The links are basically predecessor-successor relationships in the revision graph of a document. The proposed traceability approaches and tools utilize configuration management systems in order to manage this information [121, 122]. The approaches have a coarse-grained view on the evolution, because the configuration management systems see the documents as a whole and do not focus on entities inside the evolving documents. As a consequence, the tracing of entities is mostly done implicitly. It strongly relies on the names of the entities, and it makes the assumption that names do not change from one revision of a document to another. Hence, none of the approaches is applicable to the identification problem.

[1] For a thorough overview on traceability approaches we refer the reader to the surveys of Spanoudakis and Zisman [134], von Knethen and Paech [143], and Winkler and von Pilgrim [149].

2.3.2 Obtaining Traceability Links by Model Transformations

Model-driven engineering heavily uses model transformations. With a transformation one or more models are changed according to certain rule(s), e.g. the creation of get and set operations for an attribute. Transformations are often applied (semi-)automatically. The fact that a transformation applies some well-defined changes on a given input can be used to obtain traceability.

In some cases one can let the transformation create traceability information as a by-product. Jouault [66] enriches transformation rules with the creation of traceability links in the ATLAS transformation language [67]. Whenever such a transformation rule is applied, a traceability link that points from the input of the transformation to the output is created, too. A similar approach is proposed by Amar *et al.* [5], who work with EMF models [31].

A different approach has been proposed by Vanhooff *et al.* [140, 141]. They do not create one traceability link for each transformation rule, but they enable the creation of several fine-grained links for the different steps of a transformation. Other approaches that obtain traceability from transformations can be found in [11, 41, 50, 77, 115, 129, 144].

Model transformations are mostly used to transform instances of one metamodel into instances of another one; e.g. from a PIM to a PSM in MDA. Evolutionary changes, however, are mostly applied manually to the models (cf. Section 1.2). Obviously, some changes such as creation of get and set operations can easily be automated, but elementary changes such as the renaming of elements require human interaction and cannot be done by means of automated transformation. Kehrer and Ihler try to solve the problem as they propose a framework that allows us to express every possible edit operation that can be applied to a model with transformation rules [68]. With these refinement patterns each evolutionary change is done by transformation and thus traceability links capturing the evolution can be created. As a consequence, the identification over time becomes feasible as one can follow the links. If we extrapolate the idea of expressing each edit operation as a transformation, we finally apply recording, which we already discussed in the context of model repositories with tool integration (cf. Section 2.1.2.2). Recording, however, requires the development of new tools and prohibits heterogeneous tool environments. Furthermore, it becomes an exhaustive work if different modeling languages should be supported.

2.3.3 Maintaining Traceability Links

Since capturing of traceability links is a long-lasting and expensive task, approaches exist to maintain existing links. Maintenance means a proactive re-

sponse if any of the linked artifacts is changed. The link is then marked as suspect and it can be checked and corrected (mostly manually) by the user if necessary. This analysis is also referred to as consistency check or conformance analysis [101].

Maletic *et al.* have proposed an approach for maintaining traceability links between XML documents [88]. They focus on links between class models given in XML and source code which has been translated into XML. Links are given as external information using XPath expressions [152] to address elements within the documents. Whenever one of the XML documents is changed, a text-based difference algorithm is used to identify the elements (i.e. text lines in the XML file) that are affected by the change. All traceability links that point to the affected elements are marked as suspect and need to be reevaluated. The approach is generic in a way that it is applicable to nearly each kind of XML document. However, the application to models represented in XML is not trivial, because XML documents have a block-oriented structure that differs from the graph structure of models (see Section 1.4.1). Furthermore, their approach only informs about traceability links that become suspect due to changes; support for tracing evolving elements (i.e. identification) or correction of links is missing.

Similar capabilities for observing the consistency of traceability links have been presented by Munson and Nguyen with their Software Concordance framework [95, 102]. It provides versioning of software documents in XML representation. Traceability links are realized as hyperlinks, which can be versioned in the same manner as documents. Based on modification time stamps of versioned documents, they compute a conformance rating that indicates if the traceability links are potentially inconsistent. The user can then be informed about the conformance in order to maintain the links. Identification of artifacts is however based on persistent identifiers.

Murta *et al.* have proposed ArchTrace [97]. It deals with traceability links that express the implementation relation between elements of architectural models and source code. The approach focuses on the independent evolution of models and code. It supports continuous updating of links to sustain consistency whenever the model or code changes. Therefore, the approach uses custom connectors that capture changes applied in modeling tools or committed to configuration management systems. The user can define different policies of when and how links should be automatically updated. Custom connectors are used to access and change link information, which is stored either in models or in code. The level of granularity is determined by the unit of versioning of the configuration management system. Although the approach deals with model-to-code traceability, it could be adapted to model-to-model traceability. However, a major drawback is the identification of

single elements within evolving models: ArchTrace assumes the elements to have persistent identifiers.

The RT-MDD framework described by Costa and Da Silva [22] provides similar capabilities to react on changes applied to models. Their framework is based on QVT [110], but requires to hook into different tools in order to retrieve information on changes. Such hooks can hardly be applied in development environments with established tools. Similar approaches that react on events generated by editors have been proposed in [17, 18, 142].

Mäder *et al.* presented a rule-based approach to maintain traceability links within evolving models [86, 87]. Their tool, *traceMaintainer*, is able to capture change events that describe changes applied to models. Rules are used to infer development activities that have been carried out by elementary changes (i.e. patterns of changes). If end points of traceability links were affected by a change, the links can be updated automatically according to the given rules. The approach also requires to hook into tools, and it relies on persistent identifiers to identify model elements. The concept of recognizing development activities is a first step to describe the evolution that is applied to the models.

2.3.4 Recovering Traceability Links with Information Retrieval

Many valuable approaches towards recovering traceability links use information retrieval (IR) techniques [10] based on textual analysis. They have so far mainly been applied to textual documents such as specification documents or source code. We discuss their applicability to models and analyze whether they can be used to recover identification.

Antoniol *et al.* have carried out two case studies and applied probabilistic and vector space information retrieval to recover traceability links between source code and textual documents such as manuals and functional requirement specifications [6]. The IR algorithms were fed with normalized versions of the text documents and identifiers extracted from the source code. Since source code identifiers share a lot of vocabulary with documentation and requirement documents, the application of IR techniques has been advantageous. Marcus *et al.* have applied latent semantic indexing (LSI) as a third IR technique to the same case studies [89]. It has shown that LSI is in some cases even better than probabilistic or vector space models.

The ADAMS tool [23], which we already discussed in Section 2.1.2, supports the recovery of traceability links between artifacts [24]. It is based on LSI. In their case studies the recovery has not been applied to textual documents only, but to requirement documents, use cases, module descriptions, code classes, and

complete diagrams. However, the recovery is still a textual analysis, because the words of the documents and the labels in the diagrams have been used as input for the LSI.

Other IR-based approaches to recover traceability links can be found in [58, 84, 99, 157]. All approaches based on IR techniques work fine for textual documents. Also complete diagrams emerge as valid input as shown by means of the ADAMS tool. The diagrams are taken as a whole and the bulk of labels provides enough input for information retrieval. However, in case of tracing single model elements these approaches are doomed to fail, as many elements provide only few or even no text at all.

Chapter 3

The Approach in a Nutshell

In this chapter we sketch a brief summary of our approach to solve the identification problem. First we collect the requirements that are to be fulfilled by a sufficient solution. They are described in Section 3.1. Afterwards in Section 3.2 we describe the core principles of our solution by means of an example history. We also show how our approach fulfills the requirements.

3.1 Requirements

A sufficient identification approach should allow us to trace a given model element to different revisions or variants of the model. Different scenarios that require identification have been discussed in Section 1.5.1. Furthermore we have collected a set of questions regarding evolving models in Section 1.5.2. The analysis of these scenarios and questions yields the following requirements:

R1 – Suspect Selection: We must be able to select single fine-grained elements in one revision of a model, i.e. the elements to be traced across evolution. We call such an element a *suspect*. Suspects might be anonymous elements such as pseudo states or generalizations. The model revision in which we select a suspect is called the *source revision*.

R2 – Identification: For a given revision of the model (i.e. the *target revision*), we must be able to identify the element that corresponds to the suspect of the source revision at a different time. In other words, we trace the suspect from the source revision to the target revision. In the case that a suspect has been copied during evolution, the trace has to lead to all copies of that element. We call the corresponding elements in the target revision *occurrences*.

R3 – Reliability Assessment: We need to assess the reliability of an identification. In other words, we need to express how certain we can identify the suspect in the target revision. The reliability can be influenced, for instance, by the

ambiguity of selecting the right candidate and by the distance between the source revision and the target revision (i.e. the number of revisions between them).

R4 – Evolution Assessment: Besides mere identification of a suspect in other revisions, it is an important information whether the suspect has been changed in the course of time, and if so, to what extent. Hence, we require to determine the degree of evolution to which the traced element has been changed.

R5 – Occurrence Analysis: In some scenarios it is not sufficient to only identify the element that corresponds to the suspect, but additional constraints have to be fulfilled in order to consider an occurrence to *be* the other element at a different time. In the case of bug tracing, for example, we want to identify each potential occurrence unless it has been changed. The analysis must be configurable to different scenarios.

R6 – Support of Groups: Often a group of model elements builds a conceptual unit, e.g. a set of classes and their inheritance relationships. Hence, we must be able to identify groups of elements across evolution, too. This requirement is basically the extension of the above listed requirements from single suspects to groups of elements. We must also consider the case, that a group might be divided into smaller groups that evolve independently, e.g. if an inheritance relationship is removed.

R7 – Configurability: Engineering projects are unique. Each project comes with its own guidelines to modeling, and the motivation for identification can differ. Hence, the identification approach should be configurable to different scenarios, which in turn requires different measures for reliability assessment or evolution assessment.

With regard to the applicability of a tracing approach in practice, we can capture some further requirements:

R8 – Metamodel Independence: The spectrum of modeling languages ranges from generic languages, e.g. the Unified Modeling Language (UML) [107, 108], to domain-specific modeling languages (DSML), which are tailored for very particular domains [56, 93]. Examples are MATLAB/Simulink™[136] and ASCET™[40]. Furthermore, tools for the definition of arbitrary DSMLs exist [76, 79, 94]. Hence, our approach should be applicable to different types of models.

R9 – Tool Independence: We want to support collaborative work in which models are edited by distributed developers in parallel [13, 92, 126]. Hence, the approach should be independent from particular modeling tools or environments.

R10– Mineability: The need for traceability often arises in ongoing projects. Hence, our approach cannot be to record the traceability information in a modeling tool, but we should be able to mine it from histories of models that already exist, e.g. in a (file-based) configuration management system (cf. Section 1.4.3).

R11 – Extendability: The stored traceability information should be extendable if the model continuously evolves (i.e. new revisions or variants of a model are created, see Section 1.2). We should be able to incrementally compute the information for each newly added revision.

The requirement R1 is mostly fulfilled due to the identifiers used in the serialized model files, e.g. IDREFs in case of XMI [106]. If such identifiers do not exist, one can easily generate some, e.g. based on XPath expressions [152] referring to the XML element. In case of a single suspect, the identifier is just a string; for groups of elements (requirement R6), the identifier is a set of strings. We assume identifiers to be unique in the context of one model revision. In different model revisions the same identifier can of course point to different elements; otherwise we would have persistent identifiers and would not run into the identification problem (cf. Section 2.1.1). Supporting the remaining requirements for evolving models is not a trivial problem. We sketch our approach subsequently.

3.2 Our Approach by Example

We want to introduce our approach by means of the example of an evolved UML state chart model for the control of traffic lights. Figure 3.1 illustrates (a) an example version history of the model[1] and (b) depicts three revisions of it. Differences between model elements in the revisions are marked. State *On* has been renamed to *Active*, *Yellow* to *Yellow1*; the remaining elements correspond by their names. Smaller changes are encircled.

In a first step we create a representation of the version history of the model (called the *history*). It describes the different revisions of the model and their ancestor-descendant relationships (cf. Figure 3.1 (a)). The history is very fine-grained. Each model element of each revision is represented by a separate object that allows us to attach further information. The history is created by reading

[1] The numbering of the revisions is according to the numbering schema used in CVS [44].

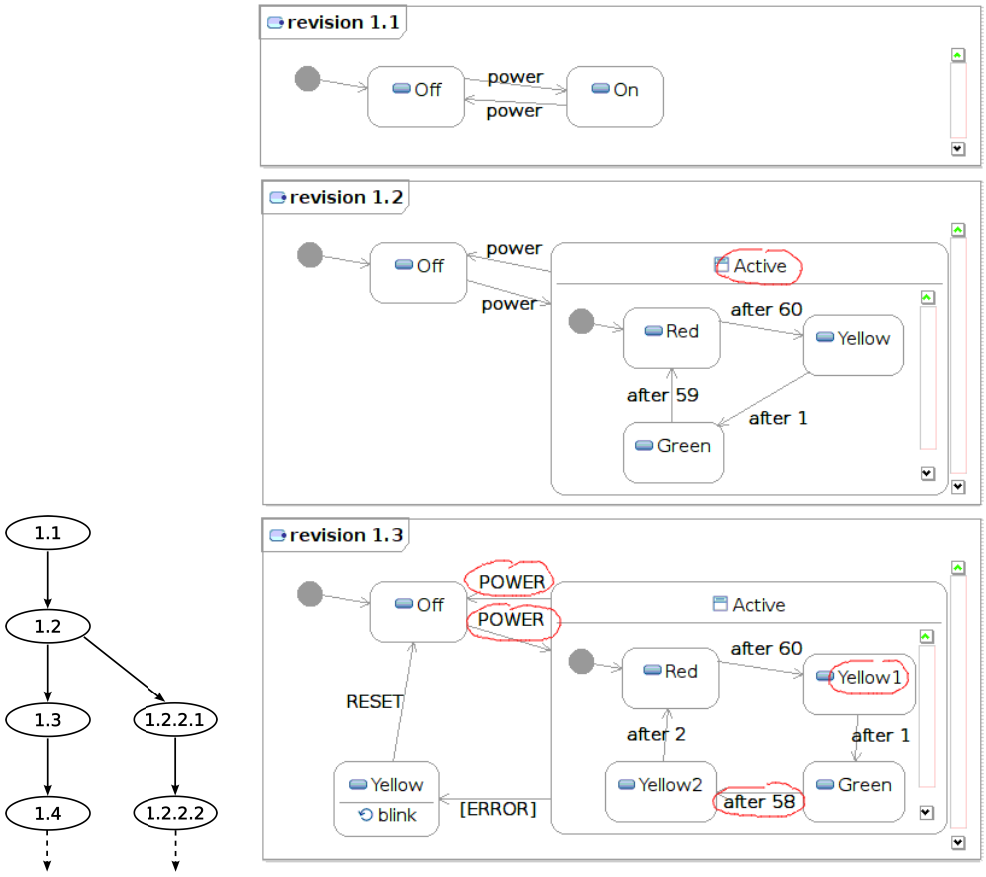


Figure 3.1: *Example version history of a model*

the different revisions of the model from a file-based configuration management system such as CVS or from a set of files on the local disk. During the import, we transform each model revision into an internal graph representation. The representation abstracts from concrete serialization formats and makes our approach independent from particular modeling tools (requirement R9). The concept of representing models as graphs is introduced in Chapter 5. The data model for histories is defined in Chapter 6.

We analyze the evolution of the model based on the beforehand created history and the graph representations of the revisions. The analysis yields traceability information and evolution information. All information is stored in the history (see Chapter 6). The core of the analysis is a pairwise comparison of the revisions according to their ancestor-descendant relationships. In simple terms, we compare each revision of the model with each successive revision of the model, either in the same branch or in parallel branches. The successors are in turn compared with

all of their successors. In the traffic lights example we compare revision 1.1 with revision 1.2, which in turn is compared to revision 1.3, and so on; revision 1.2 is also compared with revision 1.2.2.1, and so on. The comparison is executed by a flexible, similarity-based difference algorithm, called SiDiff [70, 138], that reveals correspondences between the elements of the model revisions. It does not rely on persistent identifiers and it can be configured for arbitrary graph-based model types. This makes our approach independent from particular types of models (requirement R8). The configuration further allows us to use different measures for similarity computation, thus leading to different measures for the assessment of reliability and evolution (requirement R7). Backgrounds on model comparison, the SiDiff approach, and the similarity computation are given in Chapter 4.

Based on the results of the model comparison, we create *identification links* that express the correspondence between elements of different revisions. They are a novel kind of traceability links that encode the ancestor-descendant relationship between fine-grained model elements. The identification links are the core information that is used to identify an element in another revision. Figure 3.2 shows examples of identification links in the traffic lights example. The state *On* in revision 1.1, for instance, corresponds to state *Active* in revision 1.2, which in turn corresponds to the equally named state in revision 1.3, and so on. The state *Yellow* in revision 1.2, however, does not correspond to the equally named state in revision 1.3, but to *Yellow1*. The identification links form a graph, as they connect all elements that represent the same element in different revisions including branches. An element may correspond to more than one element in the neighbored model revisions. The computation of identification links is shown in Chapter 7.

The model comparison algorithm is based on similarity heuristics. Thereby it can reveal correspondences even if the elements have changed. Since heuristics may lead to incorrect results, we need a measure to assess the reliability of the returned information. Hence, we compute a reliability value for each correspondence. It is based on the similarity of the paired elements, and the correspondences among neighbored elements. The value is assigned to the identification link, so that we can later assess the reliability of an identification (requirement R3). In the given example, we can identify the state *Off* very precisely, because it has not been changed at all and there are no ambiguities. In contrast, the state *Yellow* in revision 1.2 is identified in revision 1.3 with a lower reliability; there is a state with the same name and two states with similar names (i.e. suffixes attached). However, due to the incoming and outgoing transitions, which describe the semantics of a state, we can identify *Yellow1* to be the suspect state from revision 1.2. The computation of reliability values is described in Chapter 8.

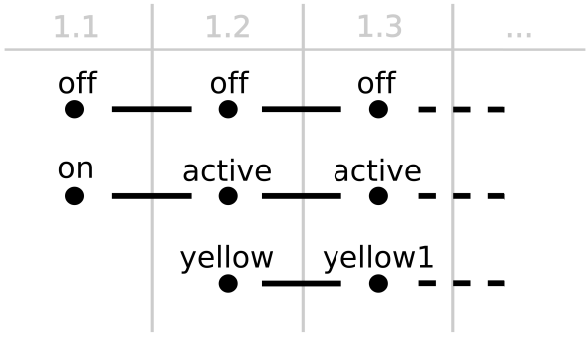


Figure 3.2: *Example identification links for the traffic lights model*

We are also able to determine the degree of evolution (requirement R4). The model comparison provides us with the difference between each pair of consecutive model revisions. In the traffic lights example, state *Yellow* has been renamed to *Yellow1*. The state *On* evolved extensively; besides renaming to *Active*, new states and transitions have been inserted. We attach the information about changes to the identification links. For a more expressive description of the evolution we also classify the changes into categories such as critical or trivial changes, and compute difference metrics that count the different types of changes. The metrics are also stored in the history. They can be aggregated over time to express the evolution in a compact set of numbers. Software product metrics that are given from outside can also be managed. Details on the types of changes and the metrics that are managed are presented in Chapter 9.

The history serves as a data warehouse. It can be queried to identify a model element over time (requirement R2). The queries essentially traverse the beforehand computed identification links to locate occurrences of the suspect in a target revision. The reporting of occurrences can be controlled by additional constraints that are validated on the elements, the identification links, or the evolution data in the history (requirement R5). Result of the queries are *traces* that point from the suspect to its occurrence in the target revision. The traces provide the user with information about the reliability of the identification. They further grant access to the changes and metrics, so that the evolution becomes more comprehensible. The identification of groups (requirement R6) is supported by separately identifying the members of a group first and analyzing the occurrences of the relationships among them afterwards. Again the reliability of a trace and the degree of evolution can be determined. We explain the possible queries on the history in Chapter 10.

Our approach is applicable to ongoing projects (requirement R10), because we import existing model revisions from a SCM or other sources. Nonetheless, we are able to continuously extend the history if a new revision is created (requirement

R11). We transform it into the internal graph representation and compare it to its predecessors. Again, we compute identification links and capture the evolution of the models elements. Therefore, we just perform the analysis as described before and add the newly computed information to the existing history.

Part II

Background & Definitions

Chapter 4

Model Comparison

In this chapter we introduce the field of model comparison. It builds the basis of our approach, since we utilize the SiDiff algorithm to compute matchings between the elements of different model revisions and to deduce the changes applied to them. We discuss the problems of model matching and model differencing in Section 4.1. An overview on existing matching algorithms is given as an excursus in Section 4.2. The excursus discusses the different types of algorithms and constitutes why we utilize the SiDiff algorithm. The SiDiff algorithm is introduced separately and in more detail in Section 4.3.

4.1 Model Matching and Model Differencing

The problem of tracing model elements over time in evolving models is very similar to the problem of model differencing. Difference computation is an important task in software configuration management. It allows us to infer the changes that have been applied to a software document [20].

In general, difference computation takes two documents as input. It is assumed that the input documents relate to each other. This relation is usually a predecessor-successor relationship in terms of versioning. In case of models we can assume the input documents to be revisions of one model.

Definition 4.1: A **model revision** is a model at a particular time. It is usually represented by a revision in a version management system or by a file. If a model evolves, we have one revision for each different state of this model.

Definition 4.2: If we compare a model revision A to another model revision B, the **(asymmetric) difference** is a description of the changes that have to be applied to revision A in order to yield revision B.

Also a symmetric difference can be defined [69]. It denotes a set of unchanged elements and two inserting transformations. However, in this thesis we only use the asymmetric definition and refer to it as *difference*.

Differences are not unique. A valid difference is always "all elements existent in revision A have been deleted, and all elements existent in revision B have been created". We are interested in a difference that better accords with the user's perception. Elements that exist in both revisions are obviously not deleted and re-inserted. They should rather be excluded from the difference, because they represent the unchanged part of a model. Hence, difference computation has to identify all elements that exist in both revisions. This task is referred to as *matching*.

In case of source code, matching is rather simple. Textual documents have a simple data type: chains of letters form a line, and if the chain of letters is changed, it forms another line (e.g. "Max is a client." vs. "Max is a customer."). Documents are sequences of lines. We can compare them by identifying the lines that exist in both revisions, e.g. by computing the longest common subsequence (LCS) [98]. The LCS algorithm compares two documents line by line. The lines of the one document are tried to be matched to the lines of the other document so that the sequence of common lines becomes maximal. Lines that differ are not matched. They are regarded as either inserted or deleted, depending on the document revision in which they exist. GNU `diff` [46] is probably the most known and used difference tool that works this way.

With respect to models, however, difference computation is more complex. Models may contain instances of arbitrary complex metaclasses that have attributes. If the attribute values are changed, the model element can still be the same (e.g. a class renamed from "Client" to "Customer"). It is not sufficient to differentiate only between inserted, deleted, and unchanged elements. We must also support the changing of elements, such as renaming or changing of other properties. Hence, the matching must not be limited to inferring equal elements. It rather has to infer *corresponding* elements, i.e. those which are the same element at different times.

Definition 4.3: Two elements are called **corresponding** if they represent the same entity in different model revisions, i.e. the elements are representatives of the same original element at different times. Corresponding elements are not necessarily equal; they might have been changed from one revision to another.

Definition 4.4: The set of correspondences between two model revisions is called a **matching**.

Given the matching between two model revisions, we can deduce a difference between these revisions. Corresponding elements that have different attributes in the different revisions lead to element changes. Corresponding elements without changes build the common subset of both model revisions, i.e. the unchanged part of the model. Elements without any correspondence are regarded as created or deleted elements.[1]

For the purpose of tracing model elements over time, we can use the matching to identify the elements of one model revision in another model revision. The computed difference information can be used to understand the evolution applied to the traced model elements.

4.2 Excursus: Approaches to Model Matching

In this section, we want to show how the matching problem can be solved. Readers who are not interested in this excursus can proceed with reading Section 4.3.

Obviously, the matching becomes trivial if we have persistent identifiers. Approaches that utilize identifiers to compare model revisions are presented in [4, 42, 90, 113]. Due to the identification problem addressed in this thesis, we do not focus on identifier-based approaches.

Although models are conceptually graphs (cf. Section 1.4.2), it is not wise to apply generic graph algorithms. Comparison of arbitrary graphs is basically the subgraph isomorphism problem, which is NP-complete [139]. Due to the tree-structure of most model types, subtree isomorphism [48] would be applicable to model matching, but it is not sufficiently efficient, either. Handling models as arbitrary graphs or trees does not take the model's syntax and semantics into account [81, 114]. The comparison on the level of textual representations, e.g. XMI files [106], is also not appropriate. The mapping of graph structures onto sequences of lines is arbitrary and changes of lines cannot be uniquely transferred to changes in the graph (cf. Section 1.4.2). It is necessary that model revisions are compared on the basis of a conceptual representation [70]. Hence, dedicated model matching approaches are required to infer correspondences. We can roughly differentiate between two classes of model matching approaches:

1. Algorithms that are adapted to specific model types. Examples are the approach by Girschick focusing on class and sequence diagrams [49] and the approach by Nejati *et al.* for matching of hierarchical state charts [100].

[1] The difference is also determined by the edit data type, i.e. the edit operations that can be executed on a model. In this example, the edit data type consists of the following operations: insert element, delete element, and change attribute. Other edit data types can be defined, too.

2. Generic algorithms that are possibly parameterized by configuration data.

Due to the fact that we do not want to restrict our tracing approach to one specific model type (see R8, page 28), we address only the second class of algorithms. It should be mentioned that the algorithms, which we subsequently discuss, are not limited to the comparison of revisions of the same model, but even independent models can be compared. Hence, in the subsequent discussion the term model can implicitly refer to a model revision.

4.2.1 Signature-Based Approaches

Signature-based approaches such as the approach of Reddy *et al.* [123] match model elements if they have the same signature. A signature is a label computed from the properties of a model element. The signature computation is deterministic so that equal elements always lead to the same signature. However, elements are not required to be equal for a matching, because only a subset of properties is encoded in a signature.

Lin *et al.* [81] use signatures in their difference tool DSMDiff for domain-specific models. Models are seen as hierarchical graphs. Signatures encode the type and the name of a node, whereby the type is divided into domain-independent type information and role information for a certain domain. Edges are assigned with similar signatures that also contain the signatures of the nodes they connect. Both models are traversed from the root to the leaves. On each level the nodes with equal signatures are matched. In case of ambiguous candidates, the signatures of edges and adjacent nodes are compared and the node with the highest number of equal edge signatures is taken. If there are still ambiguities, an arbitrary candidate is selected for the matching.

A major problem of signature-based approaches is that signatures often encode the names of elements, which prohibits the matching of renamed elements. The DSMDiff approach has furthermore the problem of top-down traversing. Models are seen as trees and they are traversed from the root to the leaves. If two elements cannot be matched, all elements belonging to the subtrees below these unmatched elements cannot match, either. Structural properties of models are only considered partially.

4.2.2 Similarity-Based Approaches

Similarity-based approaches compute similarity values for pairs of elements. The similarity values are defined by heuristics. The matching is based on the similarities, e.g. elements with the highest similarity are matched, or they are matched if

the similarity value exceeds a pre-defined threshold.

UMLDiff

The *UMLDiff* approach by Xing and Stroulia uses heuristics that are applicable to structural models [155]. The authors designed the approach to compare reverse-engineered, object-oriented designs of Java software systems. The reverse-engineered software model is therefore transformed into a directed, attributed, and typed graph. Containment relations between elements lead to edges that span a tree. Correspondences are inferred based on a pairwise similarity computation between all equally-typed elements of both models. It includes a name comparison and a structural analysis of the element. The similarity of names is given by the number of common pairs of adjacent characters. The structural similarity is given by the number of common elements in the sets of adjacent elements of the two compared elements (i.e. their intersection). Elements belong to the intersection if their names are equal or if the elements have been declared to be corresponding in an earlier iteration. Starting from the root elements of both trees, the algorithm iteratively compares all elements that are on the same logical level within the tree (i.e. they have the same distance to the root element).

Although the tool's name, UMLDiff, suggests support for the complete UML, the approach uses heuristics that are significantly tailored for OO-design. They can be applied to other structural model types, however, the application is limited. Local similarity is only given by names; other attributes cannot be considered. The computation of structural similarity weights all relations between elements equally; differentiation between different types of relations is not given. Due to the top-down traversing, the approach furthermore assumes that most changes are located in the lower levels of the tree; significant changes close to the root lead to a failed comparison.

EMFCompare

EMFCompare [30] is a recent Eclipse project, which deals with the comparison of arbitrary models built with the Eclipse Modeling Framework (EMF) [135]. It is basically a new implementation of the UMLDiff approach of Xing and Stroulia, but it overcomes some of UMLDiff's limitations. Local similarity is not given by names only, but three different similarity measures are considered: type similarity, name similarity, and value similarity. The *type similarity* measure analyzes the meta-model class of an element. As a consequence, EMFCompare can even compare elements of different types, however, this obviously raises the number of comparisons. The *name similarity* measure analyzes the name attribute of elements

similar to the UMLDiff approach. Therefore, EMFCompare searches the attribute that represents the name. The comparison is done as in UMLDiff by counting common adjacent character pairs. The *value similarity* measure includes all other local attributes of an element into the comparison. However, types of attributes are not considered; all attributes are handled as strings. The *structural similarity* measure is equal to the one of UMLDiff.

EMFCompare shares most drawbacks with UMLDiff. Although different model types are supported, it strongly focuses on structural models; behavioral models are not supported. The similarity computation is improved as it includes all attributes, however, handling them as strings is error-prone; e.g. counting common adjacent digit pairs in numbers is meaningless. The top-down traversing is still a problem if changes apply to the upper levels of the tree.

Similarity Flooding

Melnik *et al.* presented a graph matching algorithm, which is based on the idea that the similarity between a pair of vertices is determined by the similarity between its neighbors [91]. The algorithm takes two directed labeled graphs as input and creates an initial mapping between all vertices of the first graph and all vertices of the second graph. The mapping is based on a string comparison, which looks for equal prefixes and suffixes of the labels, and a similarity value is assigned to each pair of vertices. A connectivity graph is created. Vertices of the connectivity graph are the pairs of the initial mapping; they are connected by two opposing edges if the vertices of a pair share an edge in the original graphs. The final matching is computed on the connectivity graph: a flooding algorithm iteratively propagates the similarity values of the vertices (i.e. the similarity of mapped pairs) to their adjacent vertices. The similarities are equally divided onto the outgoing edges. The propagation terminates if the similarities of all vertices stabilize. Pairs with the highest similarities are selected for a match if the similarity exceeds a given threshold.

Although the approach is applicable to arbitrary directed labeled graphs, it is not well-suited for model differencing, because the graphs are seen without any semantics. Vertices have only one textual label; detailed attribution is not supported. There is no differentiation between different types of edges either. Similarities are propagated to all adjacent vertices equally. The complexity of the approach depends on the similarities of the given graphs. If the similarities do not stabilize, the termination of the propagation can only be ensured by a maximal number of iterations, which does not necessarily lead to a sufficient match result. However, in contrast to the approaches mentioned before, similarity flooding does not rely

on the tree structure of models. The sensitivity for changes to model elements of the upper levels is not given; there is no pre-defined traversal order as the algorithm works on a separate connectivity graph and similarities are flooded into all directions.

4.2.3 Rule-Based Approaches

Rule-based approaches match elements based on pre-defined rules. The rules lead to a decision whether two elements are matched or not.

The Epsilon Comparison Language

Kolovos *et al.* [74] have presented the *Epsilon Comparison Language* (ECL), which is part of the *Eclipse Epsilon* project [39]. ECL has its origin in model transformation and model merging. It enables the definition of rules that support the comparison of models of arbitrary languages. Each rule consists of two parameters that describe the types of the left element (i.e. an element of the first model) and the right element (i.e. an element of the second model) which are to be matched. A rule furthermore defines a guard, a compare part, and a conform part. The *guard* is a precondition that has to be fulfilled before comparison; it reduces the number of elements to be compared. The *compare part* does the actual comparison and results in a boolean value whether to match the elements or not. The *conform part* allows the definition of further checks to classify the matched elements into the classes of conforming elements (i.e. they are equal regarding certain properties) and non-conforming elements (i.e. they are changed). Within the rules, queries are expressed in the *Epsilon Object Language*, i.e. an extension of the Object Constraint Language (OCL). It allows the navigation and the querying on the models. For comparison, all rules are evaluated for each pair of elements of both models according to the types defined as rule parameters.

As rules are independent, the evaluation order of rules does not affect the comparison result. However, it is possible that one element matches to several other elements. In case of model differencing, multiple matches lead to ambivalent results. Furthermore, the rules decide binary and do not provide information to what extend the elements match.

The Approach by Selonen and Kettunen

Selonen and Kettunen derive structural comparison rules from a given metamodel [130]. Similar to the approach of Xing and Stroulia, their inference of correspondences relies on the name and the context of elements. The context consists of

the parent element, the mandatory neighbors (i.e. connected by a link that instantiates a metaassociation having the lower bounds of the multiplicity greater than zero), the mandatory children and grandchildren, and the mandatory neighbors of the children and grandchildren. The context is defined by the metamodel of the compared models, so that the approach can derive matching rules for arbitrary models if a respective metamodel is given. Based on these rules, each element of the first model is compared to each element of the second model. If two elements uniquely correspond (i.e. their name, type, and context are equal without any ambiguities), they are matched.

Due to the generation of metamodel based rules, the approach of Selonen and Kettunen is applicable to arbitrary types of models. It is based on equality of name and context, and a notion of similarity is not present. However, support for declaring similar elements as corresponding is necessary due to the aspect of evolution.

GenericDiff

In his most recent work, Xing proposed a new matching approach that is based on a stable-marriage algorithm and even applicable to different modeling languages [153]. Pairs of elements can be matched if so-called feasibility predicates are fulfilled. However, the publication does not provide details on the predicates. The author also writes that the current prototype implementation is very limited and that it has precision and recall values of 76% and 63%. Since these values are not sufficient for solving the identification problem, and due to the fact that a detailed description of the algorithm is not given, we do not further discuss this approach.

4.3 The SiDiff Approach

SiDiff is another approach to model matching and model differencing [70, 138]. According to the categories introduced above, the SiDiff approach is primary a similarity-based approach. However, it covers signature-based and rule-based aspects as well.

SiDiff is not a closed model differencing tool but rather an open framework and a set of libraries for building tools related to model comparison [125]. The kernel of SiDiff is a highly configurable matching algorithm based on similarities. SiDiff is applicable to all graph-based modeling languages and does not rely on persistent identifiers or unique element names.

Due to these properties, we have chosen SiDiff as the basis for the tracing of model elements over time. The computation of similarities allows us to reveal

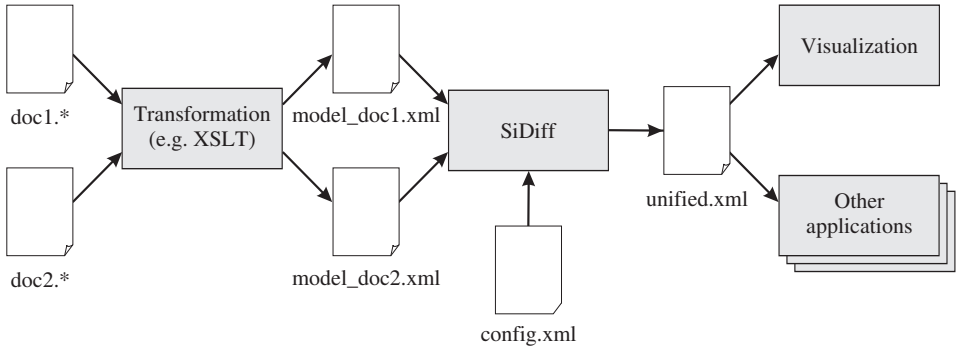


Figure 4.1: *The SiDiff pipeline (from (138))*

correspondences even between changed elements, and the high configurability of the similarity computation decouples our tracing approach from concrete modeling languages. Furthermore, the open architecture enables the extension and modification of single components to better integrate SiDiff into our approach.

4.3.1 Overview

Figure 4.1 illustrates the computation pipeline of SiDiff. SiDiff takes two graph-based models as input and transforms them into an internal format, e.g. by using XSLT transformations in case of XML files. Alternatively, the internal models can be created through an API or by implementing pre-defined interfaces if SiDiff is tightly integrated into another software. The internal models are then compared using the SiDiff kernel. Finally, the difference information (i.e. the computation result denoted as unified.xml in Figure 4.1) can be visualized or used by other applications.

The SiDiff framework has a component architecture that is based on the OSGi platform [118][2]. SiDiff can be divided into several components that provide different services. Figure 4.2 shows the major components of the SiDiff kernel. SiDiff supports different strategies to compute correspondences. For our tracing approach we focus on the *hash matcher*, which uses signatures, and the *iterative matcher*, which uses similarities. Matchers that implement different algorithms or strategies, such as identifier-based matching, exist as well, however, we skip their introduction as they will not be used for our tracing approach.

In order to match those parts of the models that have not changed, SiDiff reveals correspondences with the *HashMatcher* component, which implements an algorithm similar to the one of Wang *et al.* [145]. It computes a signature for each

[2]Here, we refer to a version of SiDiff that is newer than one introduced in the most recent publication [125].

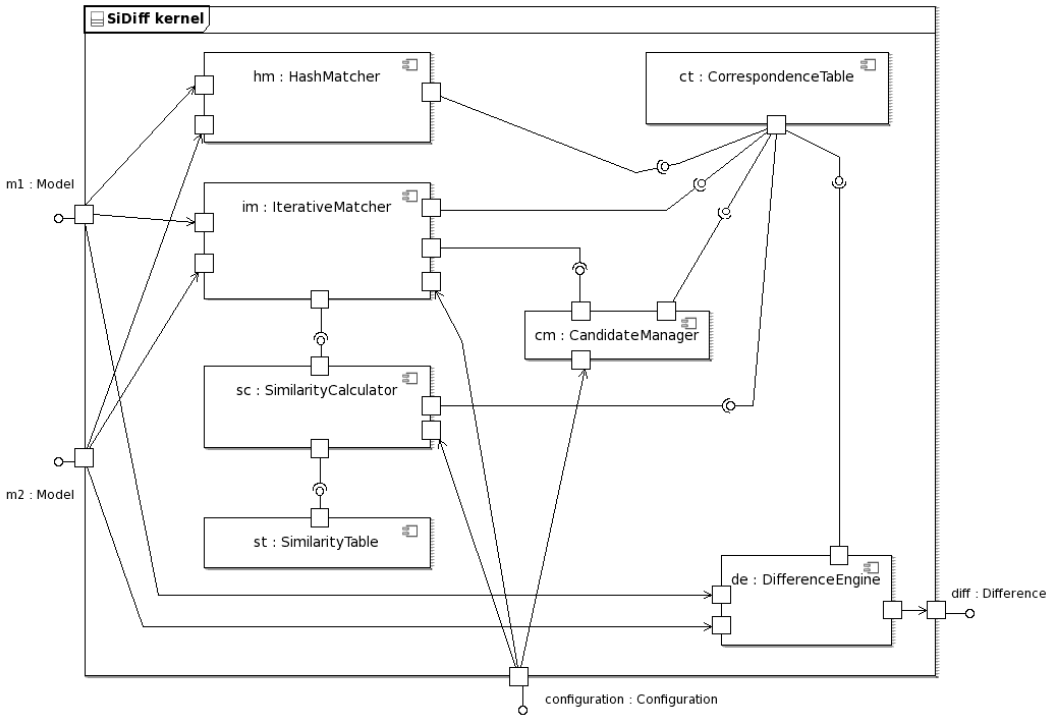


Figure 4.2: *The architecture of the SiDiff kernel*

element of the first model. The signature consists of a hash value that encodes the local attributes and nested elements and optionally a path referring to the model element. All signatures are collected in a directory. The second model is processed in a similar way. The signatures computed for the elements of the second model are compared to those stored in the directory. If two elements have unique identical signatures, they form a corresponding pair and they are matched. The hash matcher has a runtime complexity of $O(n \cdot log_2(n))$, with n being the number of elements of one model[3], so that the unchanged parts of a model can be inferred very efficiently.

All correspondences are stored in the *CorrespondenceTable*. This component is basically a table that contains one column for each model. The columns contain model elements that have a corresponding element in the other model. Correspondences are represented by rows; a row contains the model elements that are corresponding.

Those parts of the models that have changed are compared with the *IterativeMatcher* component. The iterative matcher runs a pairwise comparison of all model elements of both models that have not been matched with the hash matcher

[3]It can be assumed that models to be compared are of the same order of magnitude.

earlier. The iterative matcher uses the *SimilarityCalculator* that computes similarity values for each element pair. The value is the criterion for deciding whether a pair of model elements is corresponding. The similarities are stored in the *SimilarityTable*, which is similar to the correspondence table; it contains a column for each model and an additional column for the similarity value. Each row stores the similarity of one element pair. An element is usually a member of several pairs. Element pairs that are decided to be corresponding are stored in the correspondence table again. The computation of similarities and the matching algorithm can be parameterized. They are described in detail in Sections 4.3.2 and 4.3.3.

In order to avoid a comparison of apples and oranges, the *CandidateManager* provides a pre-selection of elements that can be compared with each other, i.e. the set of candidates for each element. Basically, two model elements are candidates for comparison if they have the same element type and neither of them has a corresponding element yet. In addition, further constraints can be defined in order to restrict the matching. If all constraints are fulfilled, a similarity value can be computed.

Based on the correspondences, the *DifferenceEngine* finally produces the difference, which is the output of the SiDiff kernel. The difference consists of a list of corresponding element pairs and detailed information about changes. Changes are classified as follows:

- An *attribute change* indicates that two corresponding elements differ in their attributes' values, e.g. a UML state that has been renamed, or the visibility of a class has been changed.

- A *reference change* indicates that the references of two corresponding elements point to different targets, e.g. an operation refers to a different return type.

- Elements that appear to change their parent elements are called *moves*. They are annotated with a reference to the other parent element, e.g. a UML class that has been moved to another package.

- Elements that have no entry in the correspondence table are considered to be *structurally different*, i.e. they have either been inserted or deleted.

Due to fact that all components of SiDiff have been realized as bundles upon the OSGi platform, we are able to reuse the components within the implementation of our tracing approach. For technical details we refer the reader to Chapter 11.

4.3.2 Similarity Computation in Detail

The similarity computation is one of the key features of SiDiff. Different than other approaches, SiDiff supports the definition of arbitrary similarity heuristics to determine whether two model elements correspond or not. The heuristics are given by configuration files that define comparison rules for each type of element.

A comparison rule defines the element properties that are relevant for the similarity of two elements of the same type. The properties are either local attributes (e.g. names) or other elements in the neighborhood. The attributes are referenced by their name, elements in the neighborhood are referenced by XPath-like expressions [152]. The comparison rules further assign a compare function to each property considered for the similarity. SiDiff provides several compare functions to analyze two equally typed properties that belong to different elements. Examples of compare functions are the comparison of attributes for equality, the comparison of strings for similarity (e.g. using the LCS algorithm [98]), the comparison of attribute values of a referenced element, and the comparison of sets of referenced elements. Each compare function returns a value between 0 and 1; a value of 0 stands for no similarity between the properties, a value of 1 expresses equality. In addition, the comparison rules assign each property with a weight indicating the relevance of the property for the similarity of two elements. The weights are chosen according to the semantics of the model type and according to what users consider a significant change. They may further depend on the application context.

The similarity between two elements is defined as the weighted arithmetic mean of the similarities of the similarity-relevant properties.

$$sim_{e_1,e_2} = \sum_{p \in P} w_p \cdot \mathrm{compare_p}(\mathrm{e_1}, \mathrm{e_2}),$$

where e_1 and e_2 are the elements to be compared, P is the set of similarity-relevant properties, w_p gives the weight of property p and $compare_p$ is the compare function for property p.

Besides the similarity-relevant properties and the compare functions applied to them, the comparison rules specify for each element type a threshold, i.e. a minimum similarity for two elements of this type to be eligible as corresponding elements. Table 4.1 shows a small excerpt of a SiDiff comparison rule for classes in UML models.

4.3.3 The Iterative Matching Algorithm in Detail

The matching algorithm processes the models in alternating bottom-up/top-down order, according to their tree-like structure. The algorithm starts from the leaves

node type = Class **threshold** = 0.5	
Criterion	Weight
Similar value for attribute *name*	0.35
Equal value for attribute *visibility*	0.05
Equal value for attribute *isAbstract*	0.05
Similar set of subelements of type *attribute*	0.20
Similar set of subelements of type *operations*	0.20
Similar elements following incoming *generalizations*	0.05
Similar elements following outgoing *generalizations*	0.05
Matched parent element	0.05

Table 4.1: *Comparison rule for classes (from (138))*

of the models and compares all elements of the same type in bottom-up direction. The similarities of all element pairs are computed as described before. Two elements are considered corresponding if their similarity exceeds the given threshold. They are matched immediately if they are not similar to any other elements. Elements that are similar to several other elements are not matched immediately because the similarities might change when further elements are compared. When all leaves have been processed, the algorithm continues with their parent elements, i.e. the bottom-up run. Each match causes the algorithm to interrupt the bottom-up run and to switch over to a top-down phase that propagates the new correspondence downwards to the children which are compared again. The initial similarities originating from the bottom-up phase can be improved since container elements or referenced elements can have been matched meanwhile. The information about matched containers enables the decision of correspondences between elements that are not allowed to be moved.[4] If their containers correspond, the element has not been moved and it can be matched. Consequently, other correspondences can be found, which are propagated top-down further on. If no further matches are found, the algorithm continues with the bottom-up phase.

If the root elements of the models are reached, the algorithm iterates over the models again starting from the leaves. Now, in the second iteration and all further iterations, similarities are recomputed and elements are matched with their most similar other element. Different than in the first iteration, it is no longer required that elements are similar to exactly one other element. The algorithm iterates as

[4] Some model elements are not allowed to be moved, i.e. they can only correspond if their containers correspond, too. An example are parameter of operations in UML model: one would only regard them as corresponding if the operations correspond. A parameter is not the same just because its name and type are equal.

long as new correspondences can be inferred. The number of iterations depends on the structure of the models.

Most models do not have a real tree structure; they contain cross references between elements, which lead to cycles. Such cycles are handled by the iterations. The similarities between elements are thus propagated through the graphs. This approach is similar to the similarity flooding algorithm [91]. It allows SiDiff to compare documents such as Petri nets, which are not tree-structured and in which the similarity of elements depends mainly on their neighborhood, and not on their compositional structure. Models that have a primary tree structure with additional cross references usually require less than 5 iterations.

Due to the pairwise comparison, the iterative matching algorithm has a runtime complexity of $O(n^2)$, where n is the number of model elements to be compared. In practice this number is limited as usually only small parts of models are changed from one revision to another. The correspondences between the unchanged model elements are efficiently inferred with the hash matcher which is executed before the iterative matching algorithm. Besides runtime reduction, the hash matcher provides a valuable set of fix points for the similarity-based comparison, i.e. if the neighborhood of elements is considered for the similarity computation.

Chapter 5

Graph Representation of Models

Model-driven engineering is not bound to particular types of models, but arbitrary modeling languages can be used. Although the different types of models vary significantly in their details, all models can conceptually be seen as graphs. We utilize this graph nature in order to keep our tracing approach independent from particular modeling languages. Furthermore, the transformation of models into a generic graph representation uncouples our approach from the technical storage formats used by different modeling tools. In Section 5.1 we introduce our definition of graphs. The transformation of models into such graphs is discussed in Section 5.2. In Section 5.3 we briefly introduce a function to query related vertices of a given vertex.

5.1 Graph Definition for Models

For a conceptual view onto models we represent models as typed, attributed, ordered, directed graphs. The graph representation is similar to *TGraphs* introduced by Ebert *et al.* [28, 29].

We define a model as a graph

$$G = (V, E, T_V, T_E),$$

where $V = \{v_1, \ldots, v_n\}$ is a set of typed vertices representing the model elements, and $E = \{e_1, \ldots, e_m\} \subseteq V \times V \times T_E$ is a set of typed edges that express the relationships among model elements. T_V is the set of all possible types of vertices, and T_E is the set of all possible types of edges. The term *model elements* refers to the entities of a model the user is actively working with. If a diagram representation of a model exists, these elements are usually represented as visual objects. Relationships are not necessarily visible in the diagram representation. An example is the type reference between a UML attribute and the class that is the type of that attribute. Edges are directed, i.e. $e_1 = (v_1, v_2, t)$ and $e_2 = (v_2, v_1, t)$ represent two

different relations. The adjacency function

$$adj : V \to E^*$$

returns the outgoing edges of a vertex as an ordered sequence. The edges are ordered according to the order of the represented relationships in the original model, if existent. The function

$$src : E \to V \quad \text{with} \quad src(e) = v \quad \text{such that} \quad v \in V \ \wedge \ e = (v, w, t) \in E$$

returns the source of an edge, and

$$tgt : E \to V \quad \text{with} \quad tgt(e) = w \quad \text{such that} \quad w \in V \ \wedge \ e = (v, w, t) \in E$$

returns the target of an edge.

It should be mentioned that model elements that are represented by edges in graphical notations (e.g. associations or transitions) are normally also represented by vertices in the graph. These elements are actively edited by the user.

As models usually consist of various types of elements and many different types of relations among them, *type* functions return a type for each vertex, and for each edge respectively:

$$type_V : V \to T_V \quad \text{and} \quad type_E : E \to T_E,$$

where T_V denotes the set of vertex types and T_E denotes the set of edge types. Both sets of types are disjoint.

$$T_V \cap T_E = \emptyset.$$

Besides their structural representation, model elements are described by several properties such as names, visibility modifiers, and others. They lead to attribution of vertices. The set of all attributes is A. The *attribute* function returns the set of attributes of a given vertex type, and the *value* function returns the value of an attribute of a certain vertex.

$$attr : T_V \to \mathcal{P}(A)$$

$$val : (V \times A) \to Z$$

It is not necessary to introduce attributes for edges, because an edge is only a reference pointing from one model element to another.

Compositional structure. Since model elements may contain other model elements, e.g. UML classes consist of attributes and operations, it is reasonable to classify some edge types as containment edge types T_C. Containment edges are directed from the vertex representing a *container* element to the vertex representing a *nested element*. They further lead to a tree-like structure of the graph. Given a

root element representing the model[1], all containment edges span a tree. Edges that are not containment edges, are reference edges T_R. The sets of containment edges and reference edges are disjoint:

$$T_E = T_C \cup T_R \quad \text{and} \quad T_C \cap T_R = \emptyset.$$

Based on the set of containment edge types T_C, we can define a *container* function that returns the container of an element:

$$cont : V \to V \quad \text{with} \quad cont(v) = \begin{cases} \emptyset & \text{if } v \text{ is the root element,} \\ w \in V \mid (w, v, t) \in E \ , \ t \in T_C & \text{otherwise.} \end{cases}$$

Due to the fact that containment edges span a tree, each model element except the root is contained by exactly one container. The container function returns $\emptyset$ for the root element. We also define a *nesting* function that returns all nested elements of a container:

$$nest : V \to \mathcal{P}(V) \quad \text{with} \quad nest(v) = \{w \in V \mid (v, w, t) \in E \ , \ t \in T_C\}.$$

According to that representation there are two ways of how to look at model elements. Simple elements, which are not containers, are represented by a single vertex. Elements that may contain other elements are represented by a subtree or by a vertex, depending on whether they actually do contain nested elements or not. In order to access container elements with their content, we define a subtree function that returns all vertices of the subtree of an element:

$$subtree : V \to \mathcal{P}(V) \quad \text{with } subtree(v) = \begin{cases} \{v\} & \text{if } nest(v) = \emptyset, \\ \{v\} \cup subtree(n) : n \in nest(v) & \text{if } nest(v) \neq \emptyset. \end{cases}$$

Hence, model elements that are containers can be represented by a set of vertices. For a uniform handling, we enable simple model elements to be represented by a set of vertices, however, this set has a cardinality of 1. Furthermore, we enable that simple and complex elements are both represented by a single vertex; nested elements, if existent, are then implicitly addressed together with their container element.

5.2 Mapping Models onto Graphs

As shown before, the graph representation of a model consists basically of a vertex for each model element and an edge for each relation. Figure 5.1 shows a snippet of a UML class model, and Figure 5.2 depicts the respective graph representation.

[1] Either a model has one distinguished root element or an artificial root can easily be created by inserting a new vertex with containment edges pointing to all elements that do not have a container.

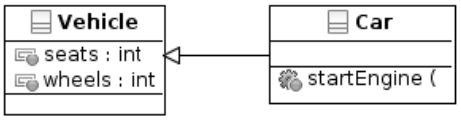


Figure 5.1: *A simple UML class model*

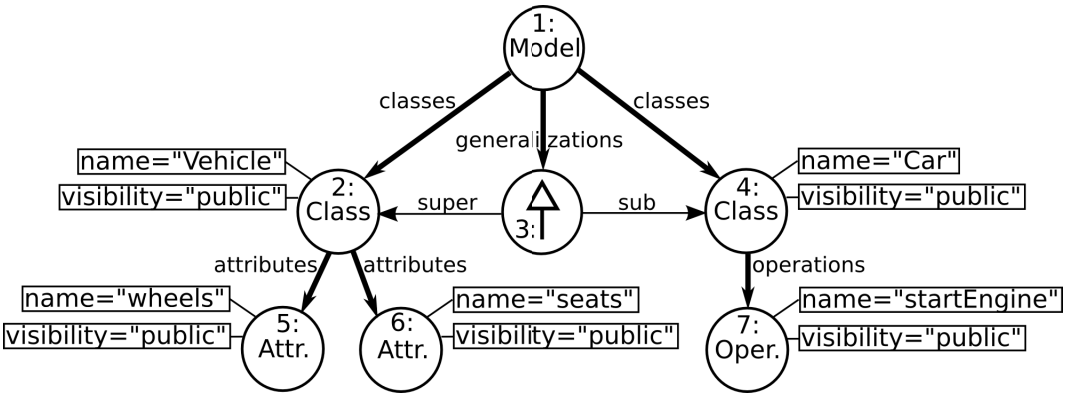


Figure 5.2: *Graph representation of the UML class model in Figure 5.1*

Vertices are represented by circles and edges by arrows. The edge type is given by labels, containment edges are shown as bold arrows. We numbered the vertices, to better refer to the figure in the following text. In general, the graph is just a different representation of a model, comparable to an abstract syntax graph. Hence, we denote the transformation of models from an external representation into our internal graph representation as *mapping*. In the figure, the representation of packages and primitive types have been omitted for clarity.

A model can be mapped onto a graph in many different ways. The *schema* of such a graph describes the different types of vertices and edges, and defines which vertices are connected by which edges. The schema can basically be deduced from the metamodel of the given model. Metaclasses become vertex types and metaassociations become edge types in the graph representation. Attributes are derived from the attributes of the metaclasses.

Metamodels are often designed only for a runtime representation of models in particular tools or for serialization. This leads to problems when it comes to computation and interpretation of differences [69]. An example is the metamodel of UML state charts, in which different kinds of pseudo states are represented by the same metaclass, and the kind of state is encoded in an attribute. In model comparison the replacement of a pseudo state would be reported as a simple attribute change, which is qualitatively equal to the renaming of a simple state. Another

alternative would be to define separate metaclasses for each pseudo state type, which leads to reporting the pseudo state replacement as a type change.

We recommend to create the schema with respect to the purpose of tracing. The set of element types should be built in a way that each type can be distinguished by a model developer; i.e. an element type differs from others in its graphical representation (in diagrams) or in its semantics. Only if the conversion between element types makes sense, it is useful to represent several kinds of elements by one vertex type that has an attribute expressing the kind. This leads to the separation of type-related and instance-related data. In UML state charts, for instance, forks and final states should have different types, while shallow history and deep history states can be represented by one vertex type "history" that has an attribute about the depth semantics.

As said before, edges in the graph representation are references between model elements, i.e. associations in the metamodel. Good examples for edges in UML class models are the reference from an operation to its return type or from a package to the classes within this package. Although some model elements express relations and are graphically represented as edges, e.g. generalizations, we advise to *not* represent them as edges in the graph. They are better represented as vertices, which are the first class citizens in our representation; especially if they may have attributes or subelements, e.g. UML associations, which contain separate elements representing the association ends. Elements that represent relationships have in turn edges pointing to those vertices whose relation they express. The set of edge types should be chosen in a way that we can differentiate between the different types of relationships among elements. In the external representation, the relationships to other model elements might be expressed indirectly by attribute values that contain the local identifier of another element, e.g. the IDREF attributes in XMI [106]. In this case, we recommend to express the relationship by an edge in our internal representation; the attribute is then not in the list of attributes of that type of model element.

The mapping of a model into graph representation has to be unambiguous and reproducible, so that the mapping of one model always results in the same graph. However, the mapping has not to be complete. We can skip model data that does not affect the semantics of a model. For example, we do not have to map attributes that contain graphical layout information such as the position of an object in a diagram if the layout is free of semantics. Another requirement for graph mappings is that they allow the translation between elements in the external model representation and vertices in the graph and vice versa. Hence, the vertices should have an attribute that serves as identifier. If the external model representation contains identifiers, we recommend to fill this attribute with the same identifier

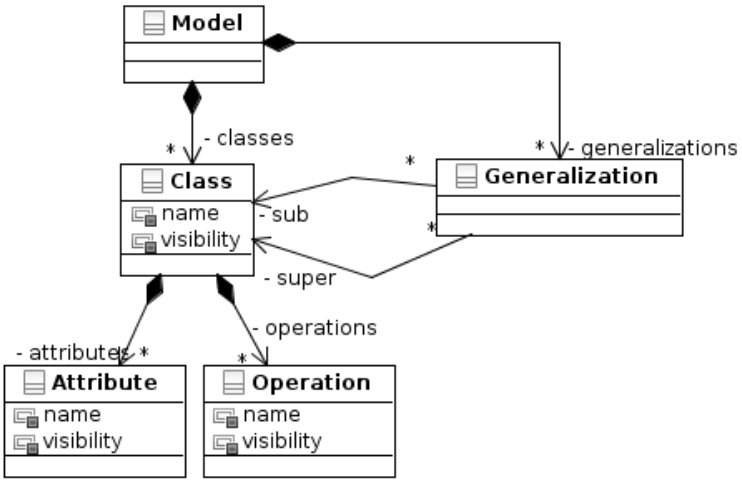


Figure 5.3: *Schema of the graph in Figure 5.2*

values. If no identifiers exists in the external model representation, we propose to use path expressions that can address the model elements.

Figure 5.3 shows an excerpt of the schema of graphs representing UML class models (cf. Figures 5.1 and 5.2). The class diagram notation of a schema is similar to the UML-based metamodeling approach of the Meta Object Facility (MOF) [105]. Vertex types are represented by classes, edge types by associations. Containment edge types are shown as composite associations. The multiplicity of associations defines the number of outgoing or incoming edges for a vertex of the respective type. All associations are navigable from source to target. The navigability, however, only describes the direction of edges of that type; our graph representation allows navigation even along incoming edges. The list of possible attributes of a vertex of a certain type is defined by the attributes of the class. Edges cannot be attributed. Inheritance can be used as short cut within a schema: Vertex types inherit all attributes and edges from their super types.

The mapping of models onto graphs makes our tracing approach applicable to all graph-based model types. Neither the concrete graph schema nor the mapping of model elements onto vertices influence the applicability of our approach. However, the mapping should follow the point of view of model developers, who will later use the traceability information, so that the changes reported by model comparison correspond with the perception of the user who modified the models.

5.3 Querying Related Vertices

In order to analyze model elements it is often necessary to capture their neighborhood or to look for model elements that stand in a particular relationship to a given element. An example is the evaluation of a compare function that checks whether related elements (e.g. the super classes) are considered corresponding (see Section 4.3.2). Hence, we must be able to evaluate queries on the graph.

A query is an ordered sequence of edge types $q = (t_1, \ldots, t_n) \in Q$, with $t_i \in T_E$, $i = 1, \ldots, n$. It can be evaluated for an arbitrary vertex $v \in V$ with the evaluation function

$$eval : (Q \times V) \rightarrow \mathcal{P}(P).$$

The function returns a set of paths. A path is an alternating sequence of vertices and edges such that the edges and the vertices connected by them form a weakly connected graph. The edges are further typed according to the query, so that the i-th edge of the path is of the i-th type of the query. The first vertex of a path is the vertex on which the query was evaluated.

$$\begin{aligned} eval(q, v) = \{\ (v_0, e_1, v_1, \ldots, e_n, v_n) \mid\ & v_0 = v \\ & \wedge\ \forall\ i = 1, \ldots, n: \quad e_i{=}(v_{i-1}, v_i, t_i)\ \vee\ e_i{=}(v_i, v_{i-1}, t_i)\ \} \end{aligned}$$

We further provide two modified evaluation functions. $eval_B$ returns only paths where consecutive edges are always different edges, i.e. we do not traverse an edge that we traversed directly before. The function $eval_C$ returns only paths that do not contain vertices twice, i.e. the paths are cycle-free. The function $query : P \rightarrow Q$ returns the query that was used to compute a given path. It is also referred to as the type of a path.

The first vertex of a path (i.e. the vertex on which the query expression has been evaluated) is called the *source vertex*. It is returned by the function

$$src_p : P \rightarrow V\ .$$

The last vertex of a path is called the *target vertex*. It is returned by the function

$$tgt_p : P \rightarrow V\ .$$

All vertices of a given path are returned by the function

$$vertices_p : P \rightarrow \mathcal{P}(V)\ .$$

The queries allow us to request model elements that stand in a particular relationship to a given model element, and they enable navigation in the models. In UML models we can for example navigate to the super class of a class or to

all subclasses. According to the mapping used in Figure 5.2, the queries are $q_{superclass} = (sub, super)$, and $q_{subclasses} = (super, sub)$ respectively. If the $eval$ function is called with vertex v_4 representing the class *Car* and the query $q_{superclass}$, it returns one path that consists of the vertices v_4, v_3, and v_2 and the edges $e_1 = (v_3, v_4, sub)$ and $e_2 = (v_3, v_2, super)$ in between. The target of the path is the super class of *Car*, which is *Vehicle*. Calling the $vertices_p$ function, we can also query the elements that form relationships, i.e. the vertex v_3 representing the generalization in the given example.

Part III

Fine-grained Traceability

Chapter 6

Modeling the History

In order to enable traceability of model elements over time we have to be aware of the evolution of the model whose elements are traced. Hence, as a core of our approach we create a history representation that describes the evolution of the model. It covers three aspects: history information, traceability information, and evolution information.

We give an overview on the history representation in Section 6.1. Subsequently we discuss the different aspects that are covered: In Section 6.2 we discuss the description of revision information. In Section 6.3 we introduce identification links that allow us to identify model elements in different revisions and thus build the basis for traceability. Finally in Section 6.4 we describe how information about evolution is stored in the history.

6.1 Overview

We assume that a model evolves over time. As a consequence, different model revisions exist. All revisions taken together form the *history* of a model.

> **Definition 6.1:** A **history** is the all-embracing description of the evolution of a model. It describes all stages of development that the model has run through.

Figure 6.1 illustrates an exemplary history. It consists of six model revisions. One revision is the root revision, i.e. the initial model. The revisions contain model elements, and they stand in an ancestor-descendant relationship, which is expressed by the gray arrows connecting the revisions. The first revision has two descendants which leads to the creation of a *branch*. The last revision of the exemplary history has two ancestors, which is called a *merger* of branches.

The history of a model can be stored in many different ways. In the most cases, the model is managed in a configuration management system, e.g. Subversion or

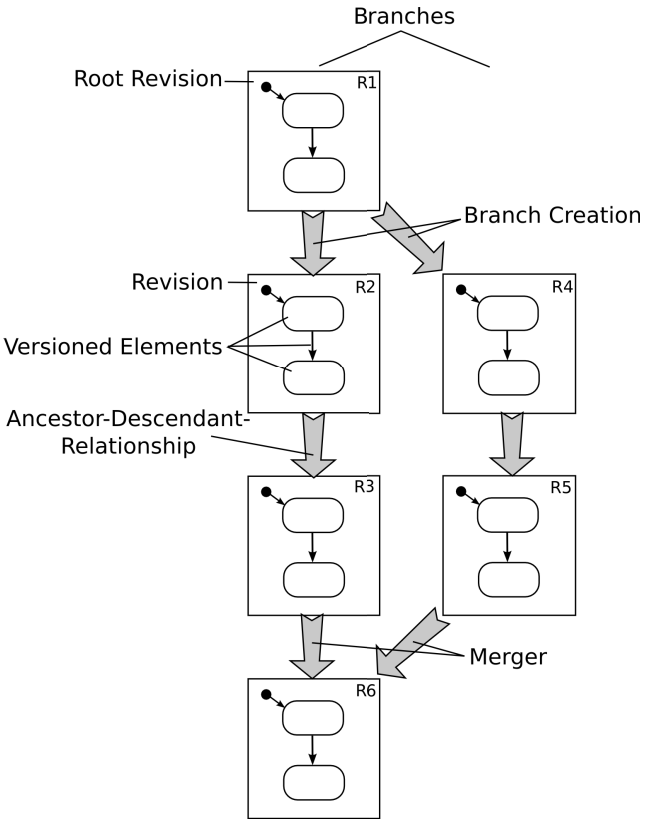


Figure 6.1: *An example history of a model*

CVS. The system stores all revisions of the model. It is also possible that the revisions are managed manually, e.g. in the file system by having one file for each revision. In the following we call the source of a history the *repository*. Since we assume file-based repositories, the stored information is very coarse-grained. We have only the model revisions given; information about the contained model elements and their evolution is not explicitly described (see Section 1.4.3).

In order to make this information explicit we create a more-detailed representation of a history. It contains information about the different revisions, their content, the evolution, and traceability information. Figure 6.2 illustrates the process of creating the history representation. We extract information about the revisions and their ancestor-descendant relationships (i.e. the revision information) from the repository. We also create a graph representation of each model revision, so that each model element that can potentially be traced is mapped onto a vertex. Different graph schemas exist for the different types of models. As described in Section 5.2, the mapping has to be unambiguous and reproducible. Based on the graph representation we can extend the revision information with information

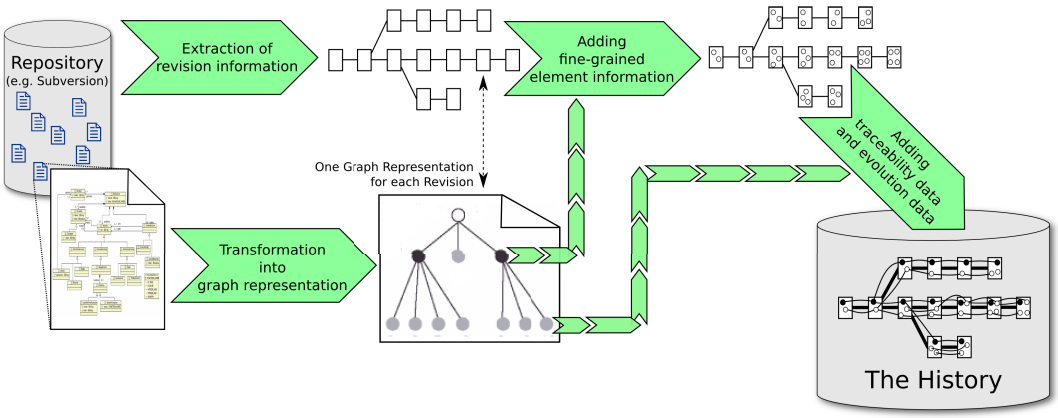


Figure 6.2: *Overview on the creation of a history*

about elements inside the revisions (i.e. the *versioned elements*).

> **Definition 6.2:** A **versioned element** is the representation of a model element at a particular time.

We create one versioned element for each vertex in the graph representation. They act as proxies for the original model elements. They do not contain the attribute data of the model elements. References to other elements (i.e. the edges in the graph representation) are not represented either. The versioned elements thus do not replace the original model elements that are stored in the repository.

After the revision information has been stored in the history representation, we add traceability information and evolution information to them. It is not required to change the data in the repository. The created representation of the history contains all information that is required to trace elements over time and to comprehend their evolution. The history is thus decoupled from concrete repository implementations. However, it does not mirror the original model revisions and can thus not replace the repository.

Mapping between repository and history. If the history is queried in order to trace elements over time or to comprehend their evolution, the queries are usually defined for the elements of the original model revisions (e.g. if the information is used in a software engineering tool). As a consequence, we have to translate between original model elements in the repository and their respective counterparts, i.e. versioned elements in the history. This translation is done by assigning an identifier to each versioned element. The identifier has to be chosen in a way that it allows us to uniquely identify the original element inside the respective model revision stored in the repository. In an ideal case, the model elements have local

identifiers that can be reused as identifiers for the versioned elements. If a model element does not have dedicated attributes for identification, we can compute a unique signature or an XPath-like expression that points to the model element and use it as identifier. Alternatively, a dictionary of internal identifiers can be created during the transformation into the graph representation. The dictionary is then the basis for mapping between original model elements and versioned elements.

Completeness of the history. It is not necessary to represent each model revision that is stored in a repository in the history. We would also be able to represent only the set of subsequent revisions that we are interested in, e.g. a single branch. However, we recommend to always create a history for the complete repository, since one cannot predict future analysis tasks. A completely represented repository further allows us to extend the history if new model revisions are added to the repository.

6.2 Representation of Revision Information

In this section we show how the revision information is stored in the history. Figure 6.3 depicts the core data model of the history. A *history* consists of revisions, and at least one revision is particularly denoted as the first revision or the root of a history. These revisions are the initial versions of the model. Usually, exactly one root revision exists. However, we accept many roots in order to support the case that a model has its origin in separately developed submodels. This can happen, for example, if different analysis models are created from different view points and these models are later merged to one global analysis model [53].

Each history is assigned with a document type. The document type defines the type of the model whose history is represented. It indirectly assigns the graph schema that is used to map the model revisions onto graph representations. Additionally, the history is assigned with a name and a description. The name is used to identify a history; it can be set to the name of the model whose evolution is represented. The description is optional and can be used for documentation.

A *revision* represents one particular model revision (see Definition 4.1). Revisions have a revision number that enables the unique identification of the revision inside the history. Again, a description can be used to document the revision. If the history of the model is managed in a configuration management system, such as CVS or Subversion, the description can be set to the commit message that has been used for creating this revision.

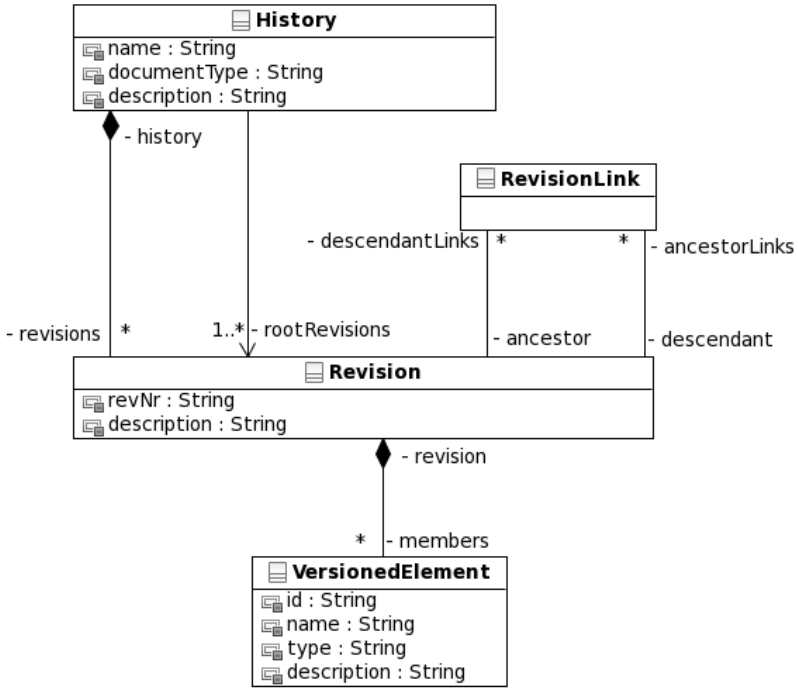


Figure 6.3: *Core data model of the history*

The ancestor-descendant relationships between single revisions are expressed by *revision links*[1]. They connect a revision with all its descendants and vice versa. Each revision can have one or more descendants and one or more ancestors.

> **Definition 6.3:** A **revision link** $rl(R_a, R_d)$ expresses the directed ancestor-descendant relationship between two revisions R_a and R_d, such that R_a is the direct ancestor of R_d.
> Each revision can have several incoming revision links that connect the direct ancestors, and several outgoing revision links that point to direct descendant revisions.

The revisions of a history that do not have ancestors (i.e. $R1$ in the example of Figure 6.1) are called the roots. Multiple descendants are given if branches are created. Multiple ancestors are given if branches are merged. For the purpose of tracing it is not necessary to distinguish between the particularities of different branches, i.e. the names, purpose, etc.; we just require the information about the different paths within the history.

[1] In the analysis model of the history the relationship could be represented by a simple association, because it has no additional attributes. However, it has been modeled as a class in order to make this relationship more explicit (see Figure 6.3).

Definition 6.4: A **revision path** $p(R_1, R_n)$ is an sequence of revisions $(R_1, \ldots, R_n)$ that are connected by revision links so that
$\forall\ R_i, i = 1, \ldots, n-1 :\ \exists\ rl(R_i, R_{i+1})$.
$\{R_{i+1}, \ldots, R_n\}$ are called descendants of R_i, while $\{R_1, \ldots, R_{i-1}\}$ are called ancestors of R_i. A revision can belong to many revision paths.

The example of Figure 6.1 contains two maximal revision paths: $p_1(R1, R6) = (R1, R2, R3,\ R6)$ and $p_2(R1, R6) = (R1, R4, R5, R6)$. Shorter paths, i.e. all subpaths of p_1 and p_2, are valid paths, too.

As a model revision contains many elements, each revision object holds a set of versioned elements. Each versioned element has an identifier that allows us to unambiguously locate this element inside the model revision. Conversely, we must be able to unambiguously locate the versioned element that represents a given model element. However, the identifier does not need to be unique across different revisions. A description can also be assigned to each versioned element. We also suggest to assign the type and, if available, the name of the versioned element in order to provide a human-readable identification of the elements. This eases the comprehension of all data that we will compute later.[2]

As said before, versioned elements act as proxies for the original elements that are actually traced. They are used to assign traceability information and evolution information to model elements without modifying the original model revisions.

6.3 Representation of Traceability Information

Once we have expressed the revisions of a model in terms of the history mentioned above, we can extend it with traceability information. Traceability can be seen from different view points:

1. We want to be able to identify a model element across its evolution. Thus, we have one identifier that uniquely identifies the element in each model revision if the element exists in that revision.

2. We want to be able to follow an element of a model revision to the corresponding element in the ancestor revision, and in the descendant revision respectively, if the element exists in that revision of the model.

Both ways lead to the same result. With a closer look we see that the existence of a globally unique identifier allows us the locate an element in the ancestor or

[2]The data is not only meant for further processing by tools, but it can also be accessed by users directly.

descendant revision, and being able to follow an element from one revision to another we can identify the element within the complete history by traversing the revision paths. However, we need the differentiation between global identification and following elements from one revision to another. The former is necessary to quickly identify elements across evolution. The latter is necessary to understand the identification and to assess how trustworthy it is. We call the existence of a globally unique identifier the *identity* and call the connection to follow an element to the ancestor or descendant revision an *identification link*.

Definition 6.5: An **identification link** $il(v_1, v_2)$ is the connection between two corresponding versioned elements v_1 and v_2 where $v_1 \in R_1$ and $v_2 \in R_2$ with $R_1 \neq R_2$ and a revision path $p(R_1, R_2)$ exists.
Identification links are directed from the element of the ancestor revision to the element of the descendant revision.

An element can have multiple identification links if branches are created or merged, or if the element has more than one corresponding elements in the other revision.[3] In accordance with Definition 4.3 the identification link does *not* imply the equality of versioned elements. The model element can rather have been changed from one revision to another.

It is worth noting that model revisions that contain elements connected by an identification link need not be connected by a revision link. This means that the revisions are not necessarily *direct* ancestor and descendant of each other, however, it is important that they are ancestor and descendant, which is expressed by the required revision path. Usually, model elements that exist in two revisions R_a and R_b do also exist in all revisions on a revision path $p(R_a, R_b)$. However, it can be that a model element is deleted in one revision and reinserted later, i.e. the deletion was revoked. In this case, revisions might exist in which the element does not exist. We call this situation a *gap*. We do not represent gaps explicitly; they are implicitly represented by identification links that connect versioned elements of revisions that do not have a direct ancestor-descendant relationship. A more precise definition of gaps and a description of how they are found will be given in Section 7.2.2.

[3] This can be the case if a model element has been copied so that the descendant revision contains duplicates.

Definition 6.6: An **identification path** $ip(v_1, v_n)$ is the set of identification links $\{l_1 = il(v_1, v_2), l_2 = il(v_2, v_3), \ldots, l_{n-1} = il(v_{n-1}, v_n)\}$ such that the versioned elements $v_1, \ldots, v_n$ and the links $l_1, \ldots, l_{n-1}$ form a weakly connected graph.

Multiple identification paths can exist between two corresponding elements if branches are created and merged between the revisions that contain the elements connected by the identification paths.

Definition 6.7: For one original model element, we define the **identity** as the set of all versioned elements representing that model element at different times and all identification links connecting them.
If two versioned elements v_a and v_b are connected by at least one identification path $ip(v_a, v_b)$, we say that v_a and v_b *can be traced*. They have the same identity.

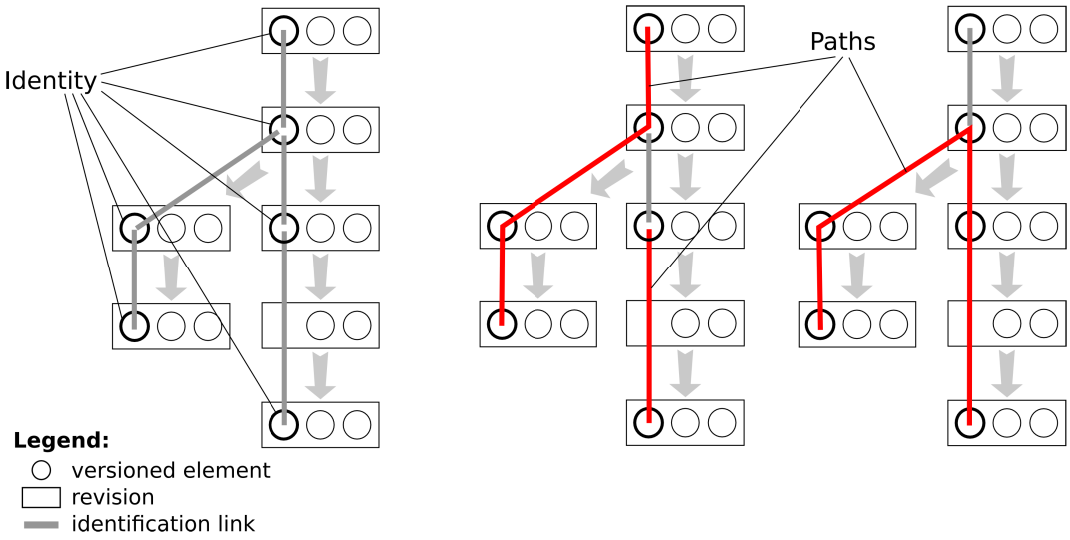


Figure 6.4: *Illustration of identification links, identity, and identification paths*

Figure 6.4 depicts an example of an identity on the left hand side. The bold circles are versioned elements that represent the same model element at different times. They are connected by identification links that are drawn as gray bold lines. On the right hand side of the figure we illustrate three examples of identification paths by the red (darker) lines.

Extension of the data model. In order to represent the traceability information, we extend our data model of Figure 6.3 by identification links and identities. The extended model is shown in Figure 6.5.

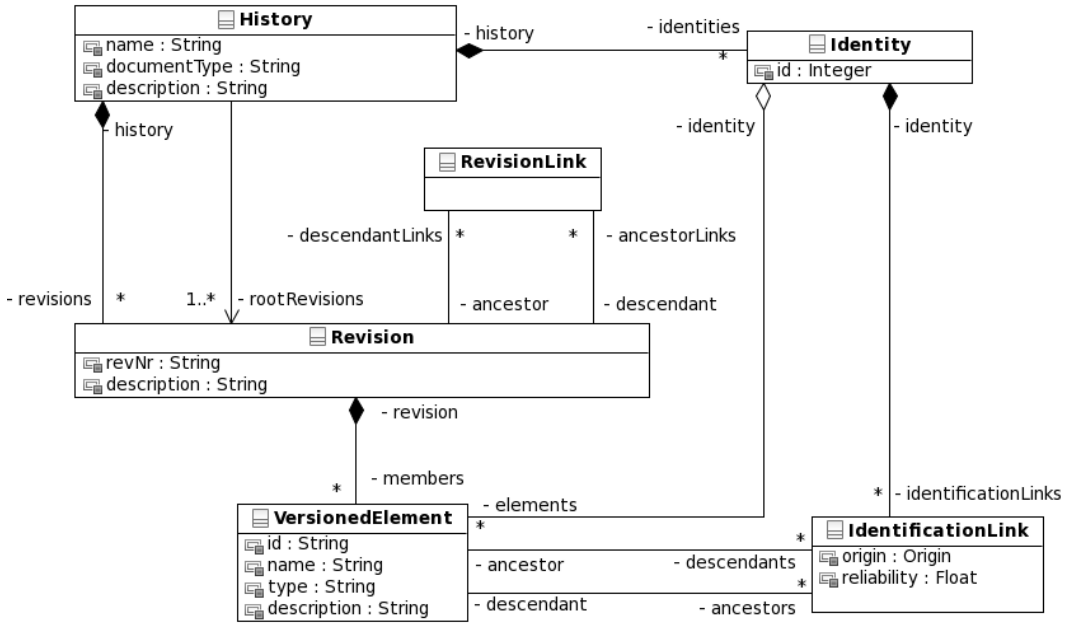


Figure 6.5: *The data model extended with traceability information*

A history contains a set of *identity* objects representing the identities. Each identity object aggregates all corresponding versioned elements of one model element. It is assigned with the surrogate id that acts as the globally unique identifier of that model element. The identity also contains the set of identification links that constitute the identity.

An *identification link* expresses the correspondence relation between two versioned elements. It is basically described by the versioned elements it connects, i.e. the element in the ancestor revision (*ancestor* for short) and the element in the descendant revision (*descendant* for short). It does not need additional identifiers, as the identification link between two versioned elements is always unique. Besides the pure correspondence information, the identification link object additionally stores quality-describing attributes, namely, the origin and the reliability. These attributes allow us to assess the identification links. The origin expresses where the identification link originates from. For example, the link can be declared manually by an expert or it can be computed by a heuristic or algorithm as we will discuss later. The reliability is a value that allows us to express how trustworthy the expressed correspondence is. Although the correspondence itself is binary such that two versioned elements do either correspond or not, another measure exists that expresses to what extent we trust in this information. Identification paths are not represented in the data model; they are derived from the identification links on demand.

6.4 Representation of Evolution Information

Since model elements can change from one revision to another, we enrich the history with information about evolution. This information allows us to better comprehend the evolution of a traced model element and it enriches the traceability information provided by the links. Therefore, we extend the data model with changes, difference metrics, software metrics, and similarity values. The result is shown in Figure 6.6.

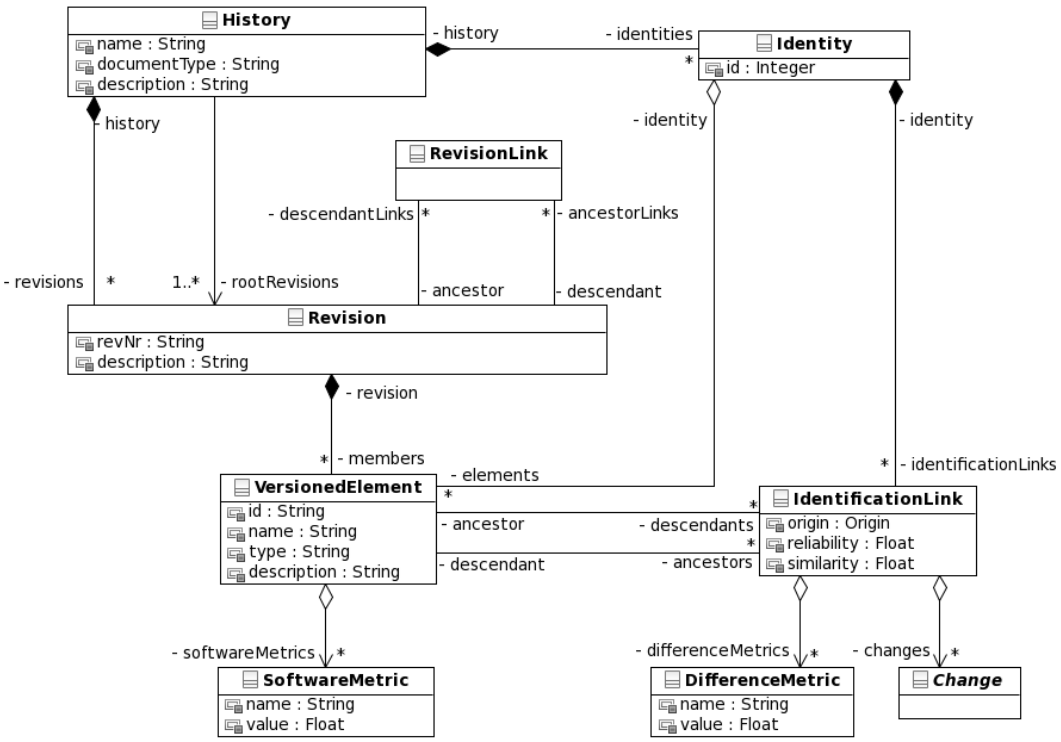


Figure 6.6: *The data model extended with evolution information*

The changes that have been applied to a model element from one revision to another are stored as *Change* objects. Each change object is assigned to an identification link. It expresses one change that has been applied to the model element represented by the source of the link (i.e. the ancestor). Different types of changes exist depending on the model type and the kind of operations that can be applied to models of that type (i.e. the edit operations). Depending on the type of change, the object holds different information that describes the change. We will discuss the types of changes in more detail in Chapter 9. All changes that are assigned to an identification link describe the evolution of the respective model element from the ancestor revision to the descendant revision. The changes are local, i.e.

they only apply to the element of the identification link they are assigned to. The changes of contained elements are assigned to their identification links. If a UML class has been renamed from one revision to another, the change is assigned to the identification link between the versioned elements of that class. If an operation of that class has been renamed, the change is assigned to the identification link of the versioned elements of that operation.

Due to the fact that a containment represents a part-of relationship, one could argue that the change of an element affects the container, too. For example, a UML class evolves if its operations are changed. In order to express such evolution we additionally assign *DifferenceMetric* objects to the identification links. A difference metric quantitatively describes the changes that have been applied to the contained elements. Each metric has a name that describes changes that are measured by the metric. An example is the difference metric *"Number of renamed operations = 5"*. It can be assigned to the class that contains the renamed operations. If it is assigned to a package, this metric expresses the number of all renamed operations of all classes inside the package. The number of difference metrics depends on the type of the model and the edit operations that can be applied to the model. The metrics are automatically derived. We discuss the difference metrics and their computation in more detail in Chapter 9.

In addition to changes and difference metrics, we can assign *SoftwareMetric* objects to versioned elements. Each object represents a software product metric that quantitatively describes some characteristic of a model element at a particular time. It has a name that describes the characteristic that is measured by the metric. An example is the software metric *"Depth in inheritance tree = 2"* for classes; it describes the position of a class in the inheritance tree. Hence, evolution is not only described by changes to model elements, but it can also be described by the changes to assigned software metrics. For example, the *"Depth in inheritance tree"* could have changed from 2 to 3, which means the another class has been inserted in the inheritance tree. The set of software metrics is not fixed, but arbitrary metrics can be defined and assigned to versioned elements. More details can be found in Chapter 9.

Finally, we extend the identification link with a *similarity* attribute. If the identification links are computed on the basis of a similarity-based model comparison approach, this attribute can be filled with the similarity value computed by the model comparison. It expresses the similarity of the two elements that are linked (cf. Section 4.3.2).

Chapter 7

Computation of Identification Links

The computation of identification links can be broken down to a pairwise comparison of subsequent model revisions. Given the exemplary history depicted in Figure 6.1, we can compute the traceability information by comparing revision $R1$ with $R2$, which in turn is compared with $R3$, and so on. Revision $R1$ is also compared with $R4$, and so on.

We process the computation incrementally whenever a new revision is added to the history. Hence, we analyze the revisions in the order of their creation, which is important because we can assume that the ancestors of each analyzed revision have been analyzed earlier. The incremental analysis allows us to apply the approach even on an actively used repository; the traceability information can thus be used in an active development environment and it can be extended easily, if a new model revision is created. The incremental behavior does thereby not impede the analysis of a complete history; we just have to ensure that we process the revisions in an order that ensures that ancestors are always analyzed before their descendants are analyzed.

The rest of this chapter is structured as follows. Section 7.1 describes the incremental computation of traceability information. In Section 7.2 we show how temporarily deleted elements are handled and we discuss the case that a correspondence cannot be found. Finally, in Section 7.3, we briefly sketch alternative approaches to retrieve the traceability information.

7.1 Computation through Pairwise Comparison

Traceability information can be computed incrementally. Whenever a new revision of a model is created, i.e. the history model is extended with a new revision object, we can compute identification links and derive further information. We assume that all ancestors of the *new revision* have been analyzed earlier, since we process the revisions in the order of their creation.

The new revision is compared with its direct ancestor by performing the model comparison of the SiDiff algorithm (see Section 4.3). The model comparison reveals the correspondences between the elements of the new revision and the elements of the ancestor revision. The correspondences lead to identification links between the elements. An exception is given by the root revisions, which do not have any ancestor. They are not analyzed directly.

Listing 7.1 shows the analysis procedure. Basically, we compare the new revision with its ancestors, and for each revealed correspondence an identification link is created. We add the link to the identity of the linked elements or create a new identity. The correspondence computation is delegated to SiDiff (line 3). The computation itself is part of the model comparison algorithm and for plain identification it can be seen as a black box, since we only need the correspondence information (lines 4–7). However, in order to enable an assessment of the identification links that we create, we also take a further look into the results and extract information how the elements have been matched. We can deduce if the elements have been matched because of their identical hash value or by the iterative matching algorithm. The *origin* of the links is set accordingly (lines 8–12). Due to the similarity heuristics of SiDiff we can also query the similarity value that is computed by SiDiff (line 13). The similarity is not necessary for the identification link and the constituted identity, but it is an indicator for the evolution of the linked element. We also store a reliability value (line 14). Gaining information on reliability requires some modifications to SiDiff. We will explain it in Section 8.1.

An identification link connects a versioned element of the new revision (i.e. the *new element*) and the corresponding element in the ancestor revision (i.e. called the *ancestor element*). Since we iterate over all direct ancestors, a versioned element can have multiple links (i.e. in the case of mergers). The links are added to the identity of the respective versioned elements (lines 15–32). By adding an identification link to an identity, the versioned elements are added implicitly. We can differentiate between five cases where an identification link is created and added to an identity. Figure 7.1 shows different snippets of an example history; each snippet illustrates one case.

1. If both elements have not been assigned to an identity yet, a new identity is created and the link is added to this identity (lines 17–20). This case applies to the elements that have been created in the ancestor revision. While comparing the ancestor with the new revision, it is the first time we handle these elements. All elements of the root revisions are handled this way, i.e. when the direct descendants of the root revisions are analyzed.

2. If the ancestor element has already an identity and the new element has not

```
1  function analyzeRevision(Revision r) {
2    for each Revision a of r.getAncestors() {
3      ModelComparison m = compare(a,r);
4      for each Correspondence c in m {
5        VersionedElement e1 = c.getElementInRevision(a);
6        VersionedElement e2 = c.getElementInRevision(r);
7        IdentificationLink l = new IdentificationLink(e1,e2);
8        if (c.isHashMatch()) {
9          l.setOrigin("HASH");
10       } else {
11         l.setOrigin("ITERATIVE");
12       }
13       l.setSimilarity(m.getSimilarity(e1,e2));
14       l.setReliability(m.getReliability(e1,e2));
15       Identity i1 = e1.getIdentity(); // identity of the ancestor element
16       Identity i2 = e2.getIdentity(); // identity of the new element
17       if (i1 == null && i2 == null) { // i.e. Case 1
18         Identity i = new Identity();
19         history.add(i);
20         i.add(l); // adding a link adds the linked elements implicitly
21       } else if (i1 != null && i2 == null) { // i.e. Case 2
22         i1.add(l);
23       } else if (i1 == null && i2 != null) { // i.e. Case 3
24         i2.add(l);
25       } else if (i1 == i2) { // i.e. Case 4
26         i1.add(l);
27       } else { // i.e. Case 5
28         Identity i = mergeIdentities(r, i1, i2);
29         if (i != null) {
30           i.add(l);
31         }
32       }
33     } // for each match
34     for each element e in a {
35       if (e.getIdentity()==null) { // ancestor element has no identity yet
36         Identity i = new Identity();
37         i.add(e);
38         history.add(i);
39       }
40     }
41   } // for each ancestor
42 ...
```

Listing 7.1: *Analysis of a revision*

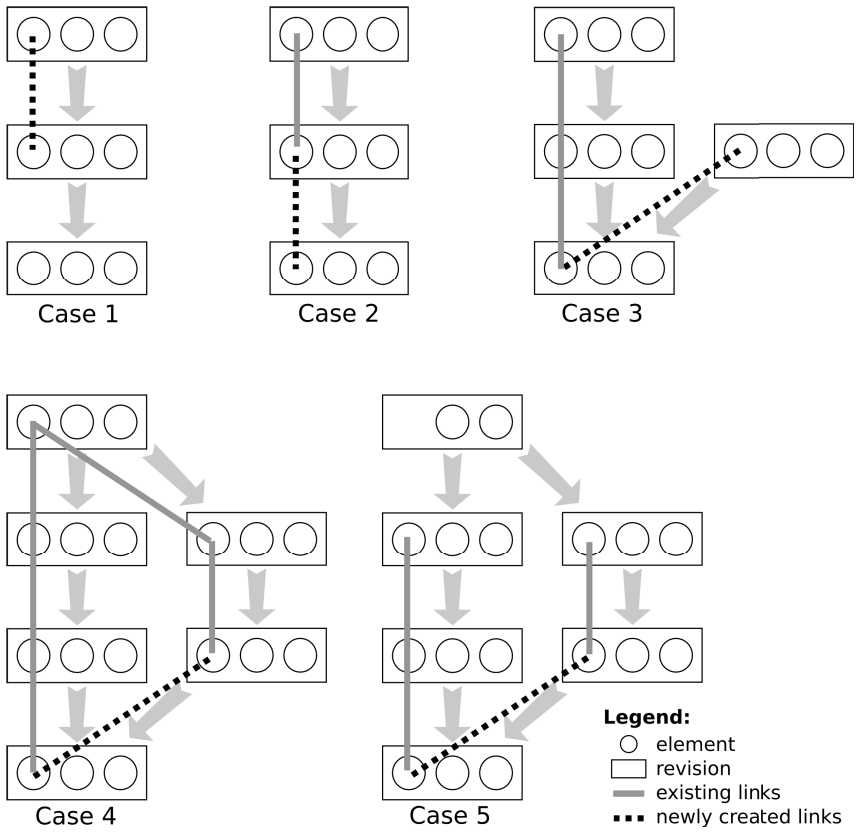


Figure 7.1: *Different cases of creating identification links*

been assigned to an identity yet, the identification link is added to the identity of the ancestor element (lines 21–22). This is the standard case: the element has been traced from its creation up to the ancestor revision, and now it can be traced further to the new revision. By adding the link to the identity of the ancestor element, the new element is implicitly assigned to the identity. The ancestor element and the new element have the same identity now.

3. If the ancestor element is not assigned to an identity yet, but the new element has already an identity, the identification link is added to the identity of the new element (lines 23–24). This case occurs if the new revision has multiple direct ancestors and the new element has been linked to an element of another ancestor revision already, i.e. in a previous iteration of the outer loop (line 2).

4. If both elements have the same identity, the identification link is simply added to this identity (lines 25–26). This case occurs if two branches are merged and the elements already existed before the creation of the branch.

5. If both elements have already been assigned to an identity but they have different identities, we try to merge the identities to a single identity. Therefore, we start a further analysis in order to check whether the identities can be merged or if they are in conflict to each other (see Section 7.1.1). If they can be merged, we do so and add the identification link to the merged identity (lines 27–32).

After processing all correspondences we create an identity for each element of the ancestor revision that has so far not been linked (lines 34–40). Thereby, we ensure that each versioned element will be assigned with an identity. However, these identities contain only one versioned element that has no correspondences in other revisions.

7.1.1 Merging Identities

In the case that the new element and the ancestor element are already assigned to identities, we have to check whether the identities can be seen to be the same identity, i.e. they can be merged. This case can only occur while merging two branches in which the same elements have been created (i.e. Case 5 in Figure 7.1). The elements did not exist before the creation of the branch, but they are similar enough to be regarded as corresponding. Hence, the identities that are to be merged have to span over two disjoint sets of revisions. If the versioned elements represent the same original model element that existed already before the branch was created, they would be assigned to the same trace (see Case 4).

If the identities span over a shared subset of revisions, i.e. both identities contain elements of a revision before the creation of the branch, we call this situation an *identity conflict*. As illustrated in Figure 7.2, merging both identities would lead to a forbidden situation; the two linked elements of revision $R1$ represent two different model elements. Accepting both identities would mean that at a different time these two model elements are represented by a single versioned element (i.e. the linked element of revision $R6$). However, according to our data model of a history, each model element is represented by a separate versioned element. As a consequence, identities that are in conflict are not merged. The new element is even removed from all identities by deleting the existing identification links. In the example of Figure 7.2, the existing link between the elements of revision $R3$ and revision $R6$ will be deleted. Finally, we have two remaining identities: one spans over the revisions $\{R1, R2, R3\}$ and the other spans over the revisions $\{R1, R4, R5\}$. The element of $R6$ is not assigned to any identity. It stays without identity until the descendant of its revision is analyzed. There it can be linked with a descendant element.

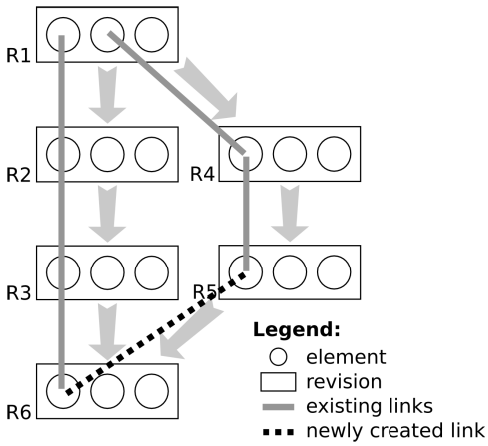


Figure 7.2: *Example of an identity conflict*

Listing 7.2 shows the analysis and merging procedure that is called if an element would become a member of two different identities. It is checked if the identities are in conflict, because they share at least one revision (lines 3–13). If they are in conflict, we remove the identification links that link to the new element (lines 14–19). If the identities are not in conflict, they are merged. Therefore, we reassign all identification links from one identity to the other. The versioned elements of the reassigned identification links are implicitly reassigned, too. The identity whose links we have reassigned is no longer needed and is deleted. The other identity is returned as the result of the merger (lines 20–28).

7.2 Handling of Breaks and Gaps

The analysis procedure described above has a significant limitation: it requires that the traced elements exist continuously and that the traced elements can continuously be connected with identification links. In other words, if the element exists in two revisions R_a and R_b, it has to exist in all revisions on the revision path $p(R_a, R_b)$ and all occurrences are connected by identification links.

This requirement can be infringed by two situations: (a) although an element exists in two subsequent revisions, we are not able to reveal a correspondence between the versioned elements, and (b) the element is deleted in one revision and re-inserted again in some later revision. Both situations are handled by an extension of our analysis procedure.

Subsequently, we discuss the two problems in more detail (Section 7.2.1 and Section 7.2.2). Afterwards, in Section 7.2.3 we describe the extension of the analysis procedure.

```
1  function mergeIdentities(Revision r, Identity i1, Identity i2) {
2    // check if there is a conflict
3    boolean conflict = false;
4    Set s = new Set();
5    for each VersionedElement e in i1 {
6      s.add(e.getRevision());
7    }
8    for each VersionedElement e in i2 {
9      if (s.contains(e.getRevision())) {
10       conflict = true;
11       break;
12     }
13   }
14   if (conflict == true) {
15     // iterate over the elements of the analyzed revision that are part of an identity
16     for each VersionedElement e in ((i1Ui2)∩r) {
17       remove e.getAncestors(); // delete the identification links of e
18     }
19     return null;
20   } else if (conflict == false) {
21     // copy all links of i2 into i1 and delete i2 afterwards
22     for each IdentificationLink l in i2 {
23       i1.add(l);
24     }
25     history.remove(i2);
26     return i1;
27   }
28 }
```

Listing 7.2: *Merging traces*

7.2.1 Breaks in the Identification Paths

Due to the heuristics of the model comparison, which builds the basis of our approach, it can happen that an element exists continuously in all subsequent revisions but we cannot connect all occurrences with identification links. We call such a situation a *break*.

Definition 7.1: Given are two revisions R_i and R_{i+1} and a model element x that exists in all revisions of the revision path $p(R_{i\ \ n}, R_{i+m})$. If the versioned elements representing x can be connected by identification links along the revision path $p(R_{i\ \ n}, R_i)$ and along the revision path $p(R_{i+1}, R_{i+m})$, but not between the revisions R_i and R_{i+1}, we call the absence of the identification link between the versioned elements of R_i and R_{i+1} a **break**.

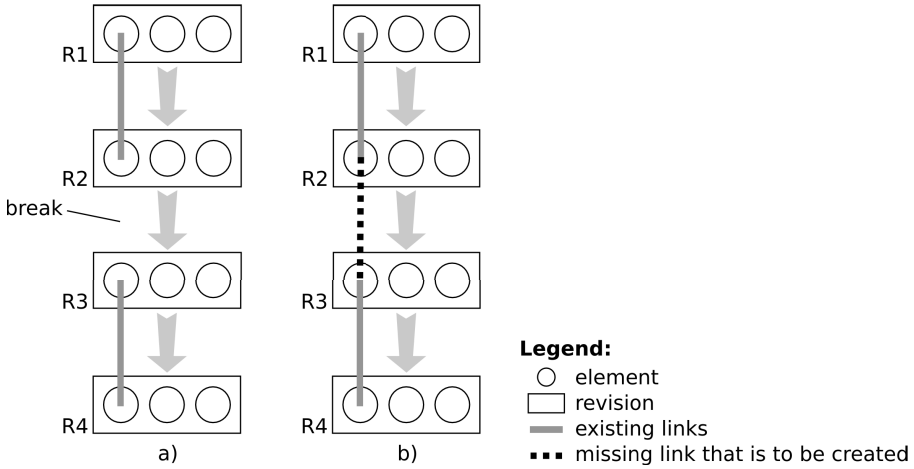


Figure 7.3: *Example of a break*

Figure 7.3 (a) illustrates a potential break. We can trace the left element of revision $R1$ to revision $R2$, but not to revision $R3$ (i.e. the analysis of revision $R3$ does not reveal an identification link). Furthermore, the element can be traced from revision $R3$ further to revision $R4$. Hence, the elements existing in revisions $R1$ and $R2$ and the element existing in revisions $R3$ and $R4$ have different identities.

If the versioned elements represent different model elements, this situation is correct. However, it might be that the versioned elements represent the same element, but we are not able to reveal the correspondence between $R2$ and $R3$. In the latter case we say that a break arised. It is caused by the heuristic of the underlying SiDiff approach, which we use to reveal correspondences. If the similarity between two elements exceeds the predefined threshold, the elements

are matched. The similarity computation can be configured very fine-grained and for models of arbitrary types. Although evaluations have attested a very low error rate for SiDiff [70], it might happen that a correspondence cannot be found. This is caused by a similarity value that does not reach the threshold (cf. Section 4.3.3). The missing correspondence leads to the break. In this case it is desirable to close the break by inserting an identification link between the versioned element of $R2$ and the versioned element of $R3$ as depicted in part (b) of Figure 7.3.

7.2.2 Deleted and Reinserted Elements

Even with the modification that handles breaks, the analysis procedure described above has still the limitation that the traced elements have to exist continuously. In other words, if the element exists in two revisions R_a and R_b, it has to exist in all revisions on the revision path $p(R_a, R_b)$. It can however happen that a model element is deleted in one revision, and in a later revision an element with the same properties is created. In this case we can argue that the deleted element and the created element are either different elements, or they are corresponding which means that this is the same element but it was not represented in the revisions between the deletion and the (re-)creation. This situation can be compared the undo function in most tools. The only difference is that the undo was performed in a later revision. The revisions between the deletion and the (re-)creation are called a *gap*.

Definition 7.2: A **gap** is the revision path $p(R_i, R_j)$ in which a model element x does *temporarily* not exist.
A gap requires that the model element exists in at least one direct ancestor of the source of the gap's revision path and in at least one direct descendant of the target of the gap's revision path. Hence, a revision path $p(R_a, R_b) \supset p(R_i, R_j)$ exists so that $x \in R_a, \ldots, R_{i-1}$ and $x \in R_{j+1}, \ldots, R_b$, but $x \notin R_i, \ldots, R_j$.

Figure 7.4 depicts an example. The left model element exists in the revisions $R1$, $R2$, $R4$, and $R5$; it does not exist in revision $R3$. The analysis procedure as described in Section 7.1 leads to the final situation shown in part (a) of the figure; the element of $R1$ is traced to $R2$, and the element of $R4$ is traced to $R5$. Hence, we would have two independent identities although the element of $R1$ corresponds to the elements in $R4$ and $R5$, too.

It is desirable to create an identification link between the element of $R2$ and the element of $R4$ as shown in part (b) of Figure 7.4. Hence, when analyzing a revision we must be able to create an identification link that points to an element of an

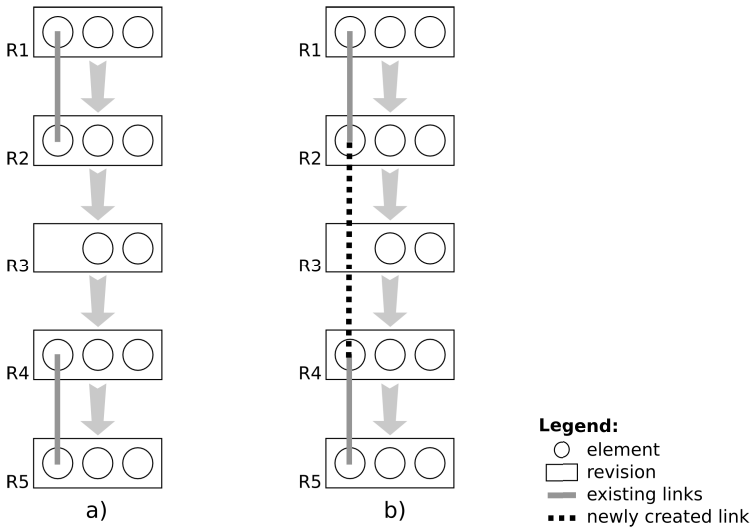


Figure 7.4: *Example of a gap*

ancestor revision that is not necessarily a direct ancestor. During the analysis of $R4$ of our example, the element of $R4$ can be assigned to the identity of the elements of $R1$ and $R2$. When analyzing revision $R5$ the identification link between the elements of $R4$ and $R5$ can be created normally, however, not a new identity is created but the existing identity is extended. We finally get one identity containing the respective elements of $R1$, $R2$, $R4$, and $R5$; the identity contains a gap in revision $R3$.

7.2.3 Extension of the Analysis Procedure

We extend our analysis procedure by a comparison of the new revision with *older ancestors*. The extension is shown in Listing 7.3. It allows us to close breaks and to create links that span over gaps. The new code is inserted after the comparison with the direct ancestors (i.e. at line 42 in Listing 7.1).

First we compute all *ancestor paths* up to a predefined length (i.e. $maxSteps + 1$) (line 42). An ancestor path is a revision path $p(A, R)$ from an ancestor revision A to the new revision R. The parameter $maxSteps$ defines how many revisions we maximally step back.[1] We iterate over the revisions of each ancestor path from the new revision into the past and do basically the same analysis as we have done for direct ancestors. We compare the revisions and for each revealed correspondence an identification link is created. The direct ancestors, which are also part of each

[1] maxSteps is a global parameter to configure the maximum distance of ancestors that are analyzed to find gaps or breaks. We recommend to set it to 2, because accidental deletion is usually revoked very soon. It can be chosen higher in order to create identification links even over larger gaps.

```
41 ...
42   for each RevisionPath p of computeAncestorPaths(r, maxSteps) {
43     for each Revision a in p {
44       if (a == r || a is direct ancestor of r)
45         continue; // skip this iteration of the loop and take next revision
46       Set s = new Set();
47       Set u1 = getElementsWithoutDescendantsOnPath(a, p);
48       Set u2 = getElementsWithoutAncestorsOnPath(r, p);
49       Set b1 = getNonContinuousDescendantsOnPath(a, p, r);
50       Set b2 = getNonContinuousAncestorsOnPath(r, p, a);
51       ModelComparison m = compareDistantRevisions(a,r,u1Ub1,u2Ub2);
52       for each Correspondence c in m {
53         VersionedElement e1 = c.getElementInRevision(a);
54         VersionedElement e2 = c.getElementInRevision(r);
55         if (e1 in u1 && e2 in u2) { // elements are so far unlinked
56           ... // create the links, i.e. is the same as in Listing 7.1, except
57               // setting the origin in order to mark links that span over a gap
58           if (c.isHashMatch())
59             l.setOrigin("GAP_HASH");
60           else
61             l.setOrigin("GAP_ITERATIVE");
62           ...
63           s.add(l);
64         } else { // at least one element has already ancestor or descendant
65           VersionedElement f1 = e1;
66           while (f1.hasDescendantOnPath(p))
67             f1 = f1.getDescendantOnPath(p);
68           VersionedElement f2 = e2;
69           while (f2.hasAncestorOnPath(p))
70             f2 = f2.getAncestorOnPath(p);
71           IdentificationLink l = new IdentificationLink(f1,f2);
72           l.setOrigin("TEMPORARY");
73           s.add(l);
74         }
75       } // for each correspondence
76       for each IdentificationLink l in s {
77         connectIfPossible(l,p);
78       }
79     } // for each ancestor revision
80   } // for each revision path
81 } // end of the analysis function
```

Listing 7.3: *Extension of the analysis procedure*

ancestor path, are skipped (line 44), since they have already been analyzed (i.e. in Listing 7.1).

The comparison of the new revision with an older ancestor requires some important modification: We are only interested in correspondences between versioned elements that are either not linked at all or that are not linked to an element in the other revision. Hence, we filter the elements that are not linked yet (lines 47–48), and we filter the elements that are not continuously linked from the one revision to the other (lines 49–50). These elements build the candidate sets. Then we call a modified comparison procedure (line 51) and create identification links according to the correspondences that have been revealed between the elements of the candidate sets. We differentiate between two cases.

If a found correspondence is between two elements that have not been linked at all yet, the creation of identification links remains the same as for correspondences between elements of subsequent revisions, however, the origin is assigned with a prefix in order to mark that the link originates from the gap analysis and that it thus spans over a gap (lines 55–62). We remember the link for further analyses (line 63).

If a found correspondence covers at least one element that is already linked with an ancestor or a descendant, we found a potential break or a potential gap. In this case we traverse along the existing links and locate the youngest descendant of the ancestor element and the oldest ancestor of the new element. Thereby we find the versioned element without descendant (f1) and the versioned element without ancestor (f2). We create a temporary identification link between them. We mark it with the origin "TEMPORARY", and we remember it for further analyses (lines 64–73).

After all correspondences have been processed we trigger a further analysis on the found links (lines 76–78).

Computation of candidate sets. We do not perform a complete comparison between the revision and its older ancestor, but we are only interested in computing correspondences between subsets of their elements. These sets are called the *candidate sets*. They are computed in lines 47–50 of Listing 7.3.

The function *getElementsWithoutDescendantsOnPath* returns those elements of the given revision (first parameter) that do not have identification links pointing to a descendant revision of the given path (second parameter), and the function *getElementsWithoutAncestorsOnPath* returns elements without identification links to ancestors revisions, respectively. The function *getNonContinuousDescendantsOnPath* returns elements of the given revision (first parameter) that have identification links to elements of descendant revisions on the given path (second

parameter), however, an identification path to elements of the other given revision (third parameter) must not exist. The function *getNonContinuousAncestorsOnPath* is the analog function that returns elements with identification links to ancestor revisions.

Figure 7.5 shows an example. The left branch represents the current ancestor path; R is the new revision being compared with the older ancestor A. In revision A the elements W and Z are not linked with any element of a descendant revision of the path. They are returned by the function *getElementsWithoutDescendantsOnPath* (i.e. $u1 = \{W_A, Z_A\}$). Element X of revision A has a descendant, however, there is no identification path to an element of revision R. Hence, it is returned by the function *getNonContinuousDescendantsOnPath* (i.e. $b1 = \{X_A\}$). In revision R the element X has not been linked at all. Z has been linked, however, it is not linked to an element of a revision on the ancestor path. Hence, the function *getElementsWithoutAncestorsOnPath* returns the set $u2 = \{X_R, Z_R\}$. Element W of revision R has an ancestor on the path, but no identification path to an element of revision A. Hence, it is returned by the function *getNonContinuousAncestorsOnPath* (i.e. $b2 = \{W_R\}$). We call the modified comparison procedure with the sets $u1 \cup b1 = \{W_A, X_A, Z_A\}$ and $u2 \cup b2 = \{W_R, X_R, Z_R\}$.

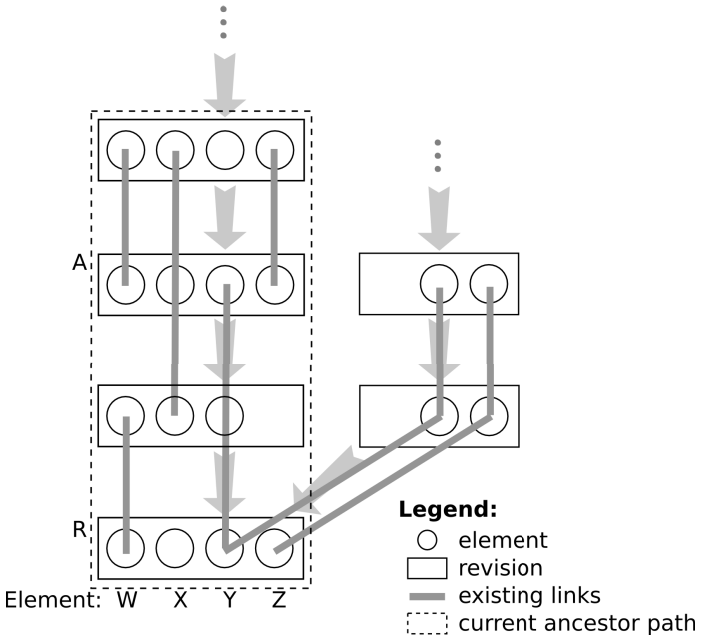


Figure 7.5: *Examples of unlinked elements*

Modified comparison procedure. In order to find correspondences only between the elements that are not linked yet or not continuously linked, we use a modi-

fied comparison procedure. It behaves similarly to the normal comparison procedure, however, it is provided with two candidate sets so that it only reveals correspondences between elements of the given sets. Furthermore we create an initial matching before we run the SiDiff algorithm. Therefore we create correspondences based on the beforehand computed identities. The existing correspondences facilitate the comparison of neighbored elements if not only local properties are analyzed to find correspondences but also information about the correspondences between neighbored elements is used (cf. Section 4.3.2).

Further analysis of found links. During the comparison of the new revision with older ancestors we might have found identification links that span over a gap and identification links that are marked to be temporary. Both kinds of links have to be analyzed in more detail.

For each identification link that spans over a gap (i.e. it does not link between elements of directly neighbored revisions) we have to check whether the element represented by the connected versioned elements has really been deleted and reinserted again. Due to similarity heuristics, which are used to reveal the correspondences, it might have happened that the correspondence was not found, because the similarity threshold has not been reached. In this case, the missing correspondence would have led to a gap that cannot be differentiated from a gap that is caused by an undone deletion. In order to avoid this kind of gaps, we try to eliminate them by inserting identification links between the elements that the model comparison has not regarded to be corresponding.

The temporary identification links can span over a break or a gap. Technically, there is no difference between these cases. They differ only in the distance between the linked elements, so that we handle them similar to the identification links that span over gaps. However, if the later analysis discovers that a temporary link does neither span over a gap nor over a break, it is deleted.

The analysis procedure for links spanning over a gap and temporary links is shown in Listing 7.4. We perform a pairwise comparison of the revisions on the path between the linked elements. This time we are not interested in the revealed correspondences but only in the similarities between the elements. We start with the ancestor element of the analyzed identification link and take the element with the highest similarity in the direct descendant revision (lines 2, 7 and 8). For this element in turn, we take the element with the highest similarity in its direct descendant revision, and so on (i.e. the loop, lines 5–17). We cancel the iteration and stop the analysis function if the element does not have a similar element in the descendant revision, or if the similarity is below a given threshold, or if additional constraints are not fulfilled (lines 9–10). The gap or the break cannot

```
1  function connectIfPossible(IdentificationLink l, RevisionPath p) {
2    VersionedElement e = l.getAncestor();
3    List s = new List();
4    Revision r1 = p.first();
5    for each Revision r2 in p\{r1} {
6      s.add(e);
7      ModelComparison m = computeSimilarities(r1,r2);
8      f = m.getMostSimilarPartnerOf(e);
9      if (f == null || m.getSimilarity(e,f) < threshold
10             || checkConstraints(e,f) == false ) {
11       if (l.getOrigin() == "TEMPORARY") {
12         remove l; // remove temporary links if their elements
13       }           // cannot be connected by another link.
14       return;
15     }
16     e = f;
17   }
18   if (e == l.getDescendant()) {
19     VersionedElement v1 = s.first();
20     for each VersionedElement v2 in s\{v1} {
21       IdentificationLink n = new IdentificationLink(v1,v2);
22       n.setSimilarity( ... );
23       n.setReliability( ... );
24       if (l.getDistance()>=2)
25         n.setOrigin("GAP_CLOSING");
26       else
27         n.setOrigin("BREAK_CLOSING");
28       l.getIdentity().add(n);
29       v1 = v2;
30     }
31     remove l;
32   }
33 }
```

Listing 7.4: *Procedure to check potential breaks*

be closed. If the analyzed link was temporary, we remove that link (lines 11–13). The constraints that we check have to be defined specifically for each type of model element. They can analyze properties on the found element or on its neighborhood. For example, they can check whether the container elements were linked, e.g. in UML models to prevent the linking of parameters without their operations being linked.

If the iteration was not canceled, we have found a similar element in the revision of the descendant element of the analyzed link. If this element *is* the descendant element of the link (line 18), we have found a chain of similar elements that can close the gap or the break. Hence, we create links between the elements of this chain (lines 19–30). We set the similarity value to the similarity between the newly linked elements and the reliability is set to $r = \frac{s^2}{o}$, where s is the similarity between the linked elements and o is the sum of the similarity values between the linked elements and all their candidates. Thus, r is a low value which expresses that this link does not provide very reliable traceability information. The origin of the new links is set to "GAP_CLOSING", unless the analyzed link was a temporary link that spanned over a break (i.e. the distance of the linked elements is 1). In the latter case we set the origin to "BREAK_CLOSING". Finally we remove the analyzed link (line 31).

The threshold used in this function is not the same as the one used by the SiDiff algorithm. Otherwise, if that threshold would be reached, we would have created a link during the normal analysis. The gap or the break would not exist. We recommend to select the threshold approximately 5–10% below the threshold of SiDiff. Thereby we can find elements that are not found by the SiDiff heuristic, but we also prohibit the matching of elements that have nothing in common.

Quality of break closing and gap closing links. It should be mentioned that the links that close gaps or breaks have to be regarded with suspicion, because they declare versioned elements to be corresponding although the heuristics of the model comparison do not. However, we advance the view that it is better to identify an element incorrectly than to state the absence of that element although it exists. An incorrectly reported identity can better be checked than an incorrectly reported absence. Thus, we assign the links with a very low reliability value and we mark them with particular origin designators. Hence, they are conspicuous if they are later used to trace a model element from one revision to another, and the traceability information can be rechecked by the user if necessary.

7.3 Alternative Approaches

Subsequently we briefly discuss alternatives to the computation procedure that we have presented above. The alternatives do not conflict with the computation procedure, but they can rather be seen as additional options that we have realized.

7.3.1 Non-Incremental Computation

We proposed the incremental computation of traceability information so that it can be used in an active development project. However, it is often the case that traceability information has to be computed for an existing history; either for analysis of older projects or because traceability becomes necessary in an ongoing project.

The computation of identification links and identities does not differ in such cases. If the existing repository provides a service to traverse through the revisions in the order of their creation, the realization is exactly the same as above. Otherwise, we completely import all revisions into our history before we start the computation with the analysis of the root revisions. We continue the computation with the analysis of the descendants of the previously analyzed revisions. If a descendant revision has further ancestors that have not been analyzed yet, we postpone this revision and continue with another descendant.

In the implementation we can profit from the initial import of the complete history. Since the revisions are analyzed directly after each other and not in separate processes, they can be kept in memory and database access can be optimized. The algorithms, however, are not optimized in this case. On a first view one could think about optimization such as multidimensional search trees for comparing all revisions at once. Such an approach has been used in [138] to provide an optimization for the similarity computation in SiDiff. Model elements of both documents are stored in a search tree and according to some predefined metrics the similar elements can be requested by a range query in logarithmic runtime. Applied to a history, we could insert all elements of all revisions into a search tree, run a range query for a suspect element, and get all corresponding elements as a result of that query. However, on a more detailed view we can see that the approach cannot be applied to our identification problem. Multidimensional search trees are designed to work in the main memory and although today's computers have a large memory they can rarely manage a complete model repository. It is furthermore a very simplifying heuristic if we reduce the complete correspondence computation to distance measures on a set of metric values.

In practice it is not necessary to reduce quality in order to improve runtime. The computation of traceability information for a complete model history is neither a

task that is executed repeatedly nor does it require user interaction. It can be performed in a nightly batch job without any problems.

7.3.2 Manual Creation

The correspondences between versioned elements of different revisions can also be created manually. The manual creation requires the expert knowledge of a user who knows the history of the analyzed model. The user can define the elements that do correspond. We enable the manual creation of identification links in order to correct the results of the automatized computation if necessary (see Section 8.2.3). It can also be used to create all traceability information. However, we cannot recommend the manual creation of all identification links of a history. Obviously, this task is very tedious, and we have revealed in one of our case studies that the manual traceability information is most likely error-prone (see Section 12.2).

7.3.3 Derivation from Identifiers

Although it is contrary to our motivation, the identification links could be derived from globally unique identifiers. The derivation is in practice not really applicable, because such identifiers come with imminent risks and they cannot be assumed in practice (see Section 2.1.1). However, if identifiers are given and if they are trustworthy, we can of course fill the history by creating respective identification links. This can be done, for example, if the model was designed by a single user with an identifier-preserving tool and the further development should be performed by several designers with different tools.

In order to derive identification links from globally unique identifiers, we have to configure the model comparison to use only the identification attribute to reveal correspondences. The model comparison and our analysis procedure presented above remains the same in order to derive the other traceability information and evolution information.

Chapter 8

Reliability and Modification of Identification Links

In the previous chapter we have shown how to compute identification links. Thus we can easily navigate from the representation of a model element in one revision to its representation in another revision. So we can identify this element within the complete history of the model. However, it is not clear whether we can trust in this information. The SiDiff algorithm, which is used to reveal correspondences between the elements of subsequent revisions, is based on heuristics. We cannot guarantee that the computed information is correct. In Section 8.1 we provide a measure that allows us to assess the reliability of the computed information. In Section 8.2 we show how the computed traceability information can be corrected by a user.

8.1 Reliability of Identification Links

Without external expert knowledge, i.e. feedback from a software engineer, we cannot decide whether a correspondence is correct or not, because we compare the model revisions on a syntactical level. Neither is it appropriate to let the user decide each single identification link. Already for the history of 10 revisions of a small class diagram with 10 classes, the user would have to make more than 1000 decisions.[1] As a consequence, we keep the computation of identification links automatized, but we enrich it with information about the reliability of a computed link. Based on that reliability information the user can be informed if the tracing results should be reviewed.

[1] If we assume that each class has 5 attributes and 5 operations in average and is connected by at least one association, we already have more than 130 model elements per revision.

Definition 8.1: The **reliability** of an identification link is a value between 0 and 1. The hypothetical value 0 expresses that we cannot assume the link to be correct, the value 1 indicates that we can trust in the information.

Obviously, a link with an assigned reliability of 0 requires manual inspection, because in this case the link should not have been created. In the other cases we use the reliability value to express how confident we were able to reveal the correspondence, i.e. if the matching given by the model comparison is clear without ambiguities. When tracing a model element from one revision to another we can assess the reliability based on the reliability of the identification links connecting the versioned elements.

8.1.1 Modification of the Model Comparison

In order to assess the reliability of a correspondence we have to know how the correspondence was computed, i.e. we need to know the circumstances that have led to the decision. Due to the similarity-based computation of SiDiff, which we use for model comparison, the similarity is at least one factor that influences the reliability. A correspondence between two very similar elements is with a high probability correct, whereas a correspondence between two very dissimilar elements should be regarded with suspicion. But there are also other criteria that influence reliability, e.g. how many elements have been a candidate of the suspicious element; matching one element out of hundreds is obviously more error-prone than matching rare elements such as the initial states of state machines.

We extended the model comparison of SiDiff to provide a reliability value for each correspondence computed by the algorithm. Therefore we extended the correspondence table to store additional information. Hence, it does not only store mappings from one element to another, but each mapping can also be assigned with a reliability value. We also extended the computation of correspondences by the computation of reliability values. We differentiate between the correspondences found by the hash matcher (i.e. the matched elements are identical in their local properties) and those found by the iterative matcher (i.e. the matchings are based on similarities, or on correspondences between other elements).

8.1.2 Reliability of Hash Matches

The hash matcher of SiDiff declares elements to be corresponding if they are equal with regard to their local properties. This heuristic is based on the assumption that large parts of a model do not change from one revision to another. Most

likely only 10% or less of a model are changed. The unchanged elements can thus be matched immediately. As described in Section 4.3, SiDiff computes a hash value for each model element, and elements with equal hash values are matched. However, it is not absolutely clear if elements with identical properties *are* the same model element at different times. An example is a UML attribute called "name" that can be found very often in class diagrams describing enterprise scenarios.

We calculate the reliability of a hash-based correspondence between two elements e_1 and e_2 as follows:

$$\begin{aligned} r_h(e_1, e_2) &= base \\ &+ a_1 \cdot path(e_1, e_2) \\ &+ a_2 \cdot container(e_1, e_2) \\ &+ a_3 \cdot (uc(e_1, e_2) - dc(e_1, e_2)) \quad . \end{aligned}$$

The constant $base$ defines a minimum reliability that we assign to elements if they have the same hash value, because the hash value is a good indicator that the elements are the same. An equal hash value does not imply that the hashed elements have the same position inside the model.[2] Hence, we also include the position of the elements in the model. The function $path$ expresses whether the elements have the same path ($path = 1$) or not ($path = 0$). The function $container$ expresses whether the container elements of the matched elements have the same hash value ($container = 1$), i.e. the matched elements are part of an unchanged subtree in the model. Obviously, a hash-based correspondence is more reliable if the neighborhood of the elements (i.e. the *context*) is corresponding, too. Thus, the last term expresses the equality of the context. We give points for neighbored elements that are unchanged (uc), i.e. they have an equal hash value, and we subtract points for elements that have differences (dc). The term is positive if we have more unchanged than changed elements in the neighborhood; it is negative if most of the neighbored elements has been changed. The definition of the context can be parameterized. For each element type, we define queries that select a type-specific context. In a UML model, for example, we define that the context of a class is given by its super classes and the classes that are connected by associations. The context of a state is given by its predecessors and successors.

It should be mentioned that the difference in the last term returns an absolute value, not a ratio. As a consequence, the reliability value returned by the formula can exceed 1. In this case it is set to 1 as the reliability must range from 0 to 1. Respectively, it is set to 0 if a negative value is calculated.

[2] The computation of hash values in SiDiff is configurable. The path to an element can be included to make the hash-based matchings more precise, however, it is usually excluded to detect model elements that have been moved.

Selection of the coefficients. The $base$ constant and the coefficients a_1, a_2 and a_3 can be adjusted to weight the influencing factors differently. They are defined separately for each type of element. If we match classes, for example, the package containing the classes is not as important as the container of states when matching states. Furthermore, we define for each type of element what the context of such an element is. It is not limited to the direct neighborhood, but it can be extended to other referenced elements. For the matching of states, for instance, we are interested in the equality of states that are connected by incoming or outgoing transitions; we are not only interested in the transitions, which are the neighbored elements in the model. The adjustment requires expert knowledge about the model type and the hash computation. The $base$ constant expresses the trustworthiness of the hash value. It should be set according to the precision and the ambiguity of the hash value. If an element type has many local properties or if it is a container for other elements with many properties, the hash value is most likely very precise and ambiguities rarely occur.[3] Accordingly, the base constant can be set to 0.5 or higher, i.e. we are by 50% sure that the equal hash value leads to a correct correspondence. If only a few properties or no contained elements are available, such as for generalizations, the base reliability should be lower; the context is more important in this case. The base constant can be higher for elements that have the character of constants, e.g. enumeration literals. They are rarely changed. The adjustment of a_1 and a_2 depends on the moveability of elements. The example is an UML attribute. The local properties are good indicators for a reliable hash, but many ambiguities can exist. If the element is still are the same position inside the document or if even the respective container elements can be matched based on their hash values, the match is more trustworthy. Coefficient a_3 should be chosen with respect to the importance and the typical size of the context. The context of a UML generalization (i.e. the super class and the subclass) is obviously very important to decide whether two generalizations correspond. Since the difference in the last term of the reliability formula does not return a ratio but an absolute value, the coefficient a_3 is not set to the portion of reliability given by context elements, but this portion is divided by the estimated typical number of context elements.

Table 8.1 shows an example definition of the reliability of hash-based correspondences between classes in UML. Since classes have many local properties such as an expressive name, we assign the hash-based correspondence with a basic reliability of 0.5. If the class has not been moved, we are 10% more sure. If even the complete package containing the class is unchanged, we are very sure (90%) that

[3] SiDiff uses all non-derived, local properties and all contained elements for the computation of the hash value.

Variable	Value
$base$	0.5
a_1	0.1
a_2	0.3
a_3	0.033
context	classes connected by associations or generalization edges

Table 8.1: *Definition of the reliability of hash-based correspondences between classes*

the class is the same. For each equal class in the context (e.g. associated classes) we add another 3.3% to the reliability. Hence, if three or more associated classes are also unchanged and no associated class has been changed, we are 100% sure that the correspondence is correct. However, if the class has been moved and is connected to several other classes that have been changed, we are not sure at all. The reliability is then even below the basic reliability $base$.

In summary, a hash-based correspondence is more reliable if the neighborhood contains further hash-based correspondences. If the contexts of the elements are different, the confidence of the correctness of the hash-based correspondence is very low. Although this is another heuristic, it allows us to very precisely estimate whether a hash-based correspondence is reliable.

8.1.3 Reliability of Iterative Matches

All model elements that have been changed from one revision to another cannot be matched by the hash matcher; they are matched by the iterative algorithm of SiDiff. In particular the similarity heuristics used in the iterative matching algorithm require further investigation regarding their reliability. The iterative matching algorithm of SiDiff computes the similarities between each pair of elements of equal type. All pairs that exceed a pre-defined similarity threshold are seen as candidates for correspondences. For the candidate with the highest similarity a correspondence is created. Four potential sources of errors exist.

1. There is always the possibility to choose the wrong element, e.g. because the correct correspondence partner has been changed significantly or it does even not exist (i.e. it has been removed from the model revision).

2. We have chosen the wrong candidate for creating the correspondence. For example, we have three candidates for an element; they have the similarities 0.9, 0.89, and 0.5. We would take the first candidate because it has the high-

est similarity, however, the second candidate has a marginal lower similarity and could be the correct correspondence. The third candidate is much more different; it is very unlikely that this is the correct correspondence.

3. The similarities that we have computed are incorrect. As mentioned in Section 4.3.2, the similarity is computed by evaluating several comparison rules that compare different properties of the model elements. Some rules compare local properties such as names of the compared model elements; their evaluation is very reliable. Other comparison rules, however, rely on the correspondences between other elements, e.g. to check whether the target of an association or transition is the same, it is checked if the targeted elements correspond. Obviously, the incorrect correspondences between referenced elements influence the reliability of the similarity of the compared elements.

4. The correspondence of the container element is wrong. Elements that are not allowed to be moved can only be matched if their containers correspond (see Section 4.3.3). The wrong correspondence of the container can thus lead to a wrong decision about the nested elements.

These sources of errors lead to the following formula to calculate the reliability of correspondences computed by the iterative matcher:

$$\begin{aligned} r_i(e_1, e_2) &= a_1 \cdot 1/log_{\sqrt{2}}(n) \\ &+ a_2 \cdot dac(e_1, e_2) \\ &+ a_3 \cdot cs(e_1, e_2) \\ &+ a_4 \cdot r_{container}(e_1, e_2) \quad . \end{aligned}$$

The first term expresses the general probability to find the correct correspondence partner. The second term deals with the selection of the partner out of the set of similar elements (i.e. out of the candidates). The third term adjusts the similarity value between the matched elements if the calculation was based on other (possibly unreliable) correspondences. The fourth term refers to the reliability of correspondences between the containers of the matched elements. Again, the terms can be weighted differently by adjusting the coefficients a_1, a_2, a_3, and a_4.

As said before, the iterative matching compares all elements of a type with each other. Hence, the number of elements of one type influences the reliability. The more elements we have in the model revision the lower is the probability that we find the right element as matching partner. This yields in a general matching probability which is $1/n$ with n being the number of elements of equal type. Since $1/n$ converges to 0 very fast, the first term would be 0 in nearly all cases. We use $1/log_{\sqrt{2}}(n)$ instead. The logarithm to base $\sqrt{2}$ behaves similar to $1/n$ for small n, but it converges slower for higher n.

The function dac describes the selection of the right candidate. It measures the distance between the similarity of the matched elements and the similarity to the best alternative candidate, and it considers the number of candidates. It can be broken down to the formula

$$\begin{aligned} dac(e_1, e_2) &= 0.7 \cdot (sim(e_1, e_2) - sim(e_1, e_2')) \\ &+ 0.3 \cdot 1/log_{\sqrt{2}}(m) \quad , \end{aligned}$$

where sim describes the similarity between two elements. e_1 and e_2 are the corresponding elements, e_2 is the most similar candidate of e_1. e_2' is the element with the second highest similarity. m is the number of candidates (i.e. the elements that have any similarity to e_1). The formula expresses the reliability of picking one of the similar candidates. We consider only the element with the second highest similarity and the *number* of candidates. This weakens the influence of the low similarities of wrong candidates. For example, we assume to have a large set of candidates such as all block elements of a subsystem in a MATLAB/Simulink™ model. All candidates except one have a similarity value of 0.3 and one candidate C has a similarity value 0.99. C is most-probably the correct candidate. However, if we consider all similarities, e.g. by computing the ratio between the candidates similarity and the sum of all similarities, the similarity of C would not stand out any more.

In order to tackle the problem that the computed similarities can be based on unreliable correspondences, we compute a *cleaned* similarity value cs. The part of the similarity that is based on other correspondences is therefore reduced according to the reliability of these correspondences. Figure 8.1 illustrates this adjustment. For each comparison rule that is evaluated to compute the similarity (cf. Section 4.3.2) we check whether it is based on other correspondences. If this is the case, we multiply the result of the comparison rule with the reliability of the correspondences used for the evaluation. Table 8.2 gives an example of the similarity adjustment of a UML class. We assume that the class is compared to another class according to the exemplary similarity configuration that was given in Table 4.1. In this example the name of the other class is slightly different, the attributes and operations are already matched with the attributes and operations of the other class, super and subclasses do not share similarities, and the containing package has not been matched, yet. The reliability of the correspondences between the attributes is in average 85%, the reliability of the correspondences between the operations is in average 95%. Due to the reliabilities of the referred correspondences we adjust the original similarity of 0.8 to the new value 0.76.

The similarity distance, dac, and the cleaned similarity, cs, are normalized to the range of possible similarities. Similarity values can range between 0 and 1, however, the real similarity values range only between x and 1, where $0 < x \leq 1$,

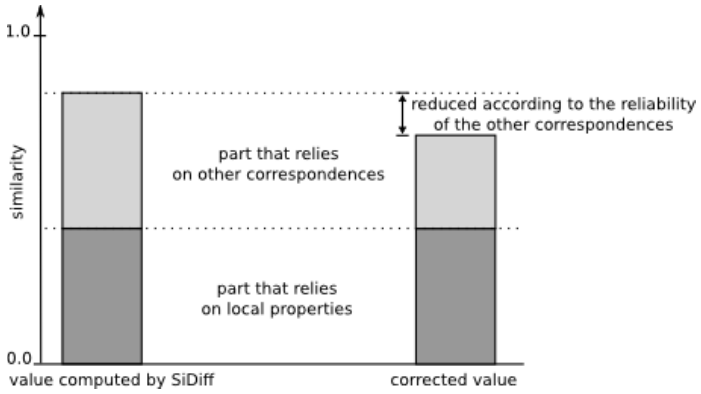


Figure 8.1: *Adjustment of the similarity*

Criterion	(W)eight	(R)esult	W*R	Relia(b)ility	W*R*b
name	0.35	0.85	0.30	1.00[1]	0.30
visibility	0.05	1.00	0.05	1.00[1]	0.05
isAbstract	0.05	1.00	0.05	1.00[1]	0.05
attributes	0.20	1.00	0.20	0.85	0.17
operations	0.20	1.00	0.20	0.95	0.19
subclasses	0.05	0.00	0.00	0.00	0.00
super classes	0.05	0.00	0.00	0.00	0.00
package	0.05	0.00	0.00	0.00	0.00
		similarity:	0.80	cleaned similarity:	0.76

[1] This criterion does not rely on other correspondences.

Table 8.2: *Detailed exemplary of a similarity adjustment*

due to the threshold used for deciding correspondences. For the calculation of dac and cs we normalize the values so that x becomes 0.01 and the value of 1 remains 1. This normalization enables an equal behavior for the different types of model elements that might have different thresholds defined in the SiDiff configuration.

Finally, with the fourth term we include the reliability of the correspondence between the containers into our calculation. For example, the correspondence between two operations is more reliable if also the classes (i.e. their containers) correspond.

Selection of the coefficients. The coefficients a_1, a_2, a_3, and a_4 are used to weight the different terms individually. Again, they are set differently for each type of model element. The selection of the coefficients requires expert knowledge about the model type and the similarity calculation of SiDiff. The sum of the coefficients

has to be 1. Otherwise, the resulting reliability would not range between 0 and 1. The general match probability can usually be weighted very low, i.e. $a_1 \leq 0.1$. It can be raised if we usually expect only a low number of elements of a given type and the element type does not have local attributes, e.g. pseudo state of kind "initial". The selection of the right candidate and the cleaned similarity are usually the most influencing factors. The setting of a_2 and a_3 should consider the similarity configuration. The more expressive the similarity is, the higher can a_2 be set. However, if the similarity is significantly based on other correspondences, e.g. if no local attributes are defined, the cleaned similarity should get more weight (i.e. a high coefficient a_3). Coefficient a_4 should be set according to the order of comparison. If it is assured by the comparison configuration that the element is only matched in top-down analysis (cf. Section 4.3.3), i.e. the container is always matched before the element, the reliability of the container element can be considered. This is usually the case if the element is not allowed to be moved (i.e. the container has to remain the same). If the container is not necessarily matched before the element, we recommend to set $a_4 = 0$.

The reliability values enable an assessment of the trustworthiness of the identification links that are derived from the correspondences of the model comparison algorithm. Whenever we trace a model element, we can appraise the quality of the information we are provided with.

8.2 Manual Editing of Identification Links

Although our approach provides very reliable results (see evaluation in Section 12.1), the user might be caused to change the computed traceability information. For instance, if the reliability value is very low for an identification link, the user can review this link, and adjust the reliability or remove the link. Furthermore, the user is enabled to create links manually if elements do not correspond although they should. Subsequently, we describe the different scenarios that can occur, and we show how they are handled.

8.2.1 Removing a Versioned Element from an Identity

A scenario that can be fixed very easily is that an element has been identified incorrectly, i.e. it does not correspond to the other elements of that identity. This can happen, for instance, if our approach has closed a gap by creating correspondences that do not exceed the similarity threshold. In order to fix this incorrect identity, the element has to be removed from the identity. Therefore the incom-

ing and outgoing identification links of that particular element are removed and new identification links between the ancestors and the descendants are created. If multiple ancestors or descendants exist (i.e. due to branching and merging), new links are created between each possible pair of an ancestor and a descendant. The removed element is assigned to a new identity. Figure 8.2 depicts this situation. The similarity value of the new identification link has to be computed be comparing the ancestor element with the descendant element. The reliability value is set to the minimum of the reliabilities of the removed links. The origin is set to "RELINKED".

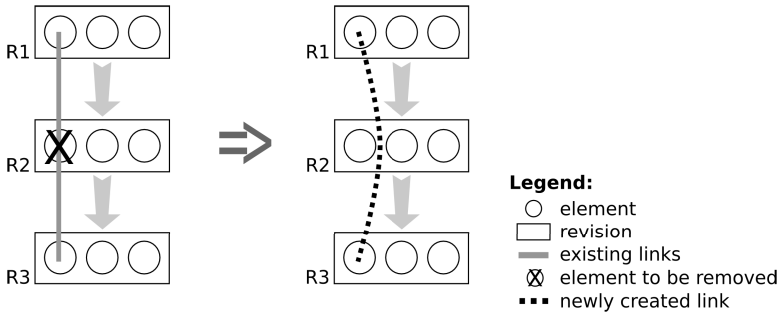


Figure 8.2: *Removing an element from an identity*

8.2.2 Removing an Identification Link

Besides removing a versioned element from an identity, the user might not agree with single identification links. This is the case if only the ancestor *or* the descendant of an element should not correspond. Hence, the identification link should be removed. In contrast to the previous case where an element has been removed from the identity while its ancestors and descendants keep their identity, the identity has now to be split into two identities. Figure 8.3 depicts this situation. The ancestor element of the removed link as well as all further ancestors keep their identity. The descendant element and also all of its descendants will be assigned to a new identity.

When removing an identification link, the ancestor element and the descendant element of that link might additionally be connected via another path of identification links. This is the case if links of a branch are to be removed (see Figure 8.4). Here, the link is simply removed without splitting the identity. As a consequence, the ancestor element and the descendant element can still be traced to each other, however, the reliability is usually much lower.

The manual removing of identification links has to be performed very carefully. It can be used to split identities in a way that the elements of a branch will be

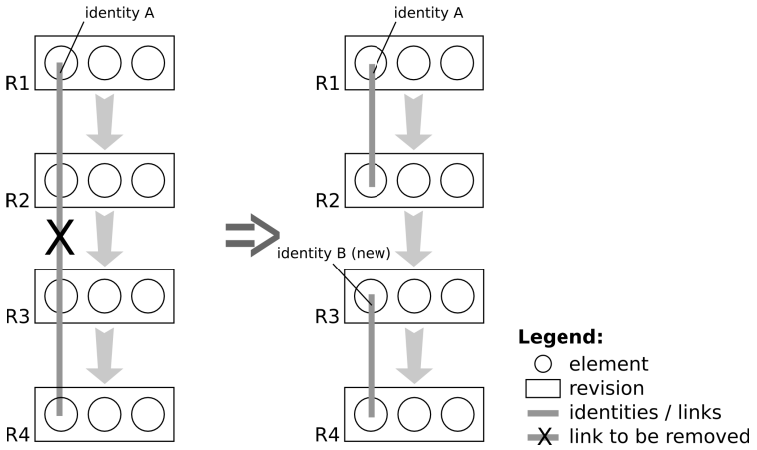


Figure 8.3: *Removing an identification link*

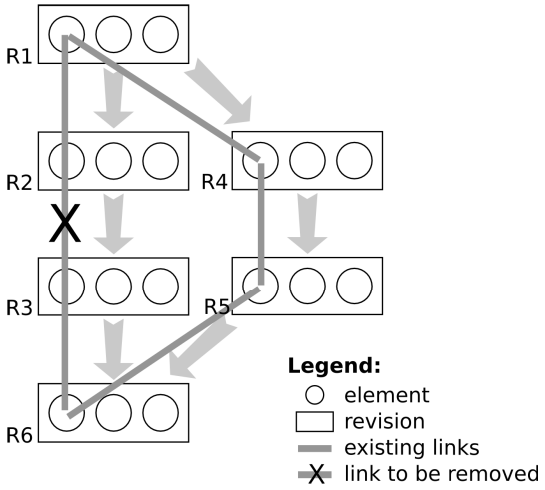


Figure 8.4: *Removing an identification link without splitting the identity*

assigned to a new identity. Therefore, the identification links along the merging revision link (if existent) should always be removed before the identification links that are along the revision link creating the branch.

8.2.3 Creating an Identification Link

It is also possible to manually extend the computed traceability information with new identification links. The user can create an identification link between two elements on one revision path if they are not yet connected to each other by another identification link. The elements are either of subsequent revisions or there is a gap in between, because the newly linked element has been been deleted and

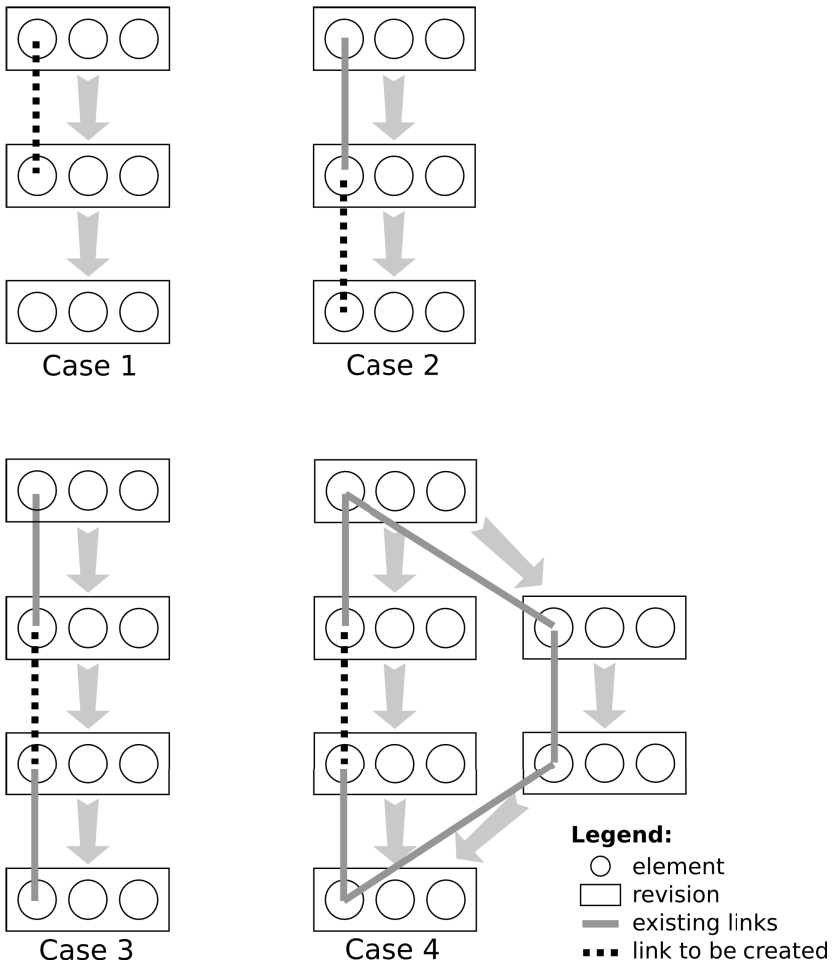


Figure 8.5: *Different cases of creating identification links manually*

re-inserted. We differentiate four different cases (see Figure 8.5).

In the simplest case (Case 1) the elements are not linked to other elements. Here, we can create the link and reassign the descendant element of the new link to the identity of the ancestor element. If one of the elements to be linked (either the ancestor or the descendant) has already a link (Case 2), we create the new link and reassign the element, which was not previously linked, to the identity of the element that had already a link. If both elements are linked already, we have to check whether the elements have different identities. In such case (Case 3) we create the new identification link and merge the identities. All elements of the identity of the descendant element will then be reassigned to the identity of the ancestor element. This is equal to the merging of identities described in Section 7.1.1. The identities of the ancestor and the descendant must not conflict. If both elements that are to be linked have already the same identity (Case 4), we

only have to create the additional identification link.

In all cases the newly created link is assigned with a similarity value that is computed by comparing the linked elements. The user is asked to provide the reliability value, because he can best describe how sure he is with his interaction. The origin of the created identification link is set to "MANUAL".

8.2.4 Changing Reliabilities

Besides removing or creating additional identification links, the user can also improve the traceability information by changing the reliability of a link. If the user has reviewed a link, he can attest that the correspondence expressed by the link is correct. Thus, the reliability can be set to 1 according to the belief of the user. The origin of the link is set to "MANUAL". Hence, in later analysis we can be sure, that the link is correct. The user can also set the reliability to a smaller value. He can thus express that he is not totally sure with his decision.

Chapter 9

Computing Evolution Information

Besides the pure traceability information, we are also interested in the evolution of the analyzed model. In other words, it is not enough to know only the identity of an element, but the knowledge about its evolution is important, too. Hence, the user should be provided with the changes that have been applied to the model elements over time. Therefore, our history representation contains also information about the evolution of model elements. In this chapter we describe how this information is computed. Section 9.1 discusses the computation of software metrics. The inference of changes is described in Section 9.2. In Section 9.3 we introduce difference metrics that are a new concept to express differences in numbers. Finally, in Section 9.4 we briefly sketch the recomputation of difference metrics and changes if identification links are modified.

9.1 Software Metrics

As shown in Section 6.4, we can assign versioned elements with software metrics. A software metric quantitatively describes some characteristic of a model element in a revision (i.e. at a particular time). Each metric is stored as a tuple that consists of a name or description of the metric, and a value.

Many software metrics basically count specific elements in a model revision, i.e. *counting metrics*. For example, in UML the metric “Number of classes” counts all classes either in the model or in the package depending on the context of the metric. The term context refers to the element to which the metric is attached. Counting metrics can easily be computed on the internal graph representation of the model revision, e.g. by traversing over the tree that is spanned by the containment edges. Some counting metrics take even certain properties of the elements into account. The metric “Number of public attributes”, for example, counts only those attributes contained by a UML class that have their visibility set to public. It is not required that metrics count only elements that are reachable via contain-

ment edges. Other edge types or even paths can be traversed, too, e.g. in order to compute the number of subclasses of a given class.

Other metrics could do a further analysis of the model elements or if the model elements represent some real entities, they can be derived from there. An example is a reverse-engineered class model of a program: we could enrich the model elements with metrics such as the McCabe metric or the Halstead metric, which express the complexity of the code that has been reverse-engineered [83].

Our history representation can handle arbitrary software metrics regardless their origin. The computation of software metrics itself is not part of our traceability approach. We assume the software metrics to be given from outside, e.g. by triggering an extra tool that returns the metrics. Furthermore, our approach is even applicable if no software metrics are defined. The metrics are only used to assist the user in the comprehension of the evolution of traced model elements. They are optional.

9.2 Inference of Changes

Since we compute identification links with the help of a model comparison approach, we can easily deduce the changes that have been applied to a model element from one revision to another. The knowledge about the changes helps the developer to comprehend the identification if the suspect is different in the target revision. The available types of changes do strongly rely on the type of the analyzed model and the edit operations assigned to that type (i.e. the operations that can be applied to models of that type). Due to our internal graph representation, the model comparison provides us with four types of changes (see Section 4.3): An *attribute change* denotes the change of the value of an attribute of an element, e.g. if the name or the visibility of a UML class has been changed. *Reference changes* indicate that a reference of an element (i.e. an edge in the internal graph representation) points to a different target element (vertex) than before, e.g. the edge expressing the return type of an operation points now to a different class. *Moves* express that an element has been moved from one container to another, i.e. the containment reference has changed. *Structural changes* mark the elements that have been inserted or deleted, i.e. the elements do not exist in the other revision.

We store the changes of an element as objects assigned to the identification link that represents the correspondence of the changed element (see Section 6.4). We do not store information about structural changes. That information is implicitly given by the absence of identification links. If an element has been inserted it has no incoming identification link. A deleted element has no outgoing link respectively. The other types of changes are represented by objects of the types depicted

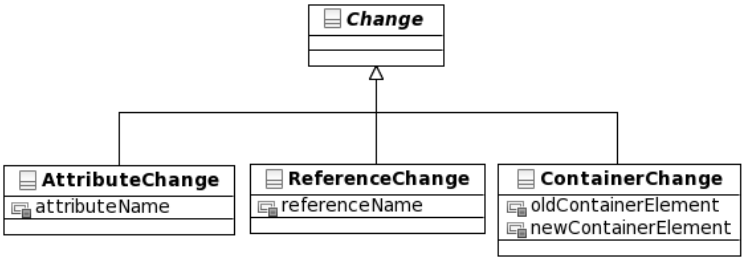


Figure 9.1: *Different types of changes that can be assigned to an identification link*

in Figure 9.1. We create an *AttributeChange* object for each concrete change of an attribute value. It holds the name of the attribute that has been changed. For each reference that points to another target now, we create a *ReferenceChange* object. The name of the reference (i.e. declaring its type) is assigned to the change object. An identification link can be assigned with several *AttributeChange* and *ReferenceChange* objects if more than one property of the linked element has been changed. An object of type *ContainerChange* expresses a move and contains pointers to the old and the new container of the moved element. Since an element has only one container, there is one *ContainerChange* object at maximum.

It is furthermore possible that additional types of changes are defined for a certain model type. Technically, these changes are then subclasses of the abstract class “Change” so that they can be handled by our approach in equal manner.

9.3 Difference Metrics: Measuring the Changes

Many model elements are containers for other model elements. We can argue in UML state machines for example that the change of a simple state leads also to a change of the composite state that contains the simple state. If we assign all changes to an identification link that do somehow change the model element represented by the link, the user will drown in a plethora of information. As a consequence, we differentiate between different levels of changes. Furthermore, if we later trace a model element from one revision to another, the user should be provided with all changes that have been applied to the traced element. The number of changes can explode then and the user is neither able to get an overview nor to capture the relevant changes.

In order to tackle that problem we define *difference metrics* that map changes of model elements onto numerical values. These metrics enable us to count, aggregate, or classify changes according to their relevance. The metrics map all changes onto a data set that is much smaller and better comprehensible than the complete

list of single changes. The set of available difference metrics is determined by the type of the analyzed model and the available types of changes. Difference metrics are assigned to the identification links. Each identification link does not contain only a list of changes that have been directly applied to the model element whose evolution is expressed by the link, but also a list of difference metrics that count these changes and all changes that have been applied to the contained elements.

We should mention that our approach differs from the approach of Demeyer et al. [26]. They have introduced the term *change metrics* in context of detecting software refactorings. However, they compute object-oriented metrics of parts of two versions of a software system and inspect the differences between the produced values subsequently. This combination of metrics and software changes is significantly different from our work, as they measure changes of software metrics resulting from software changes. We do not compute the difference between metrics, but we compute metrics on differences.

Requirements. The computation of difference metrics is independent from any traceability scenario. It is based on the tree-like graph representation of two model revisions. It requires a correspondence table that denotes pairs of corresponding model elements and a description of the difference between the two revisions. The correspondence table may contain information about similarities between corresponding elements, and the difference description should provide information about changed attributes, changed references, moved elements, and structural changes (i.e. elements that exist only in one revision). We compute the difference metrics during the computation of identification links, because it already includes a model comparison that provides us with a correspondence table, similarities, and the difference. The computation is thus based on the types of changes that are returned by SiDiff; it is not based on the change objects assigned to identification links.

Types of difference metrics. In terms of computation of metrics we can differentiate between two groups of metrics. The first group can be seen as generic metrics. All metrics of that group are derived directly from the list of types in the graph schema, i.e. the metamodel. The second group, so-called significance metrics, take model semantics into account. They require additional information that exceeds standard metamodel information based on knowledge of the application domain of the analyzed model.

9.3.1 Generic Metrics

Metrics that count changed elements. Obviously, we can count for each element the different types of changes that have been applied to that element. We call the resulting metrics *atomic*. Atomic metrics can be computed for each element and for each type of change. An example in UML is the number of attribute changes of an element of type operation; other examples are the number of reference changes of an element of type transition, or the number of structural changes of an element of type activity. These metrics usually return rather small values. Frequently, the value can only be 0 or 1 depending on whether the element was changed or not.

Due to the small values it is reasonable to accumulate atomic metrics in the corresponding container elements, e.g. to retrieve the number of changed operations contained in a given class. On this basis we define four kinds of *accumulated metrics*:

Changed children. This metric counts all nested elements that are assigned with any difference of the type *Attribute change* or *Reference change*, e.g. the number of operations changed in a class.

Inserted children. For a given element (i.e. the current element), this metric counts all nested elements that have been inserted into the current element. I.e. the nested elements that are assigned with a *Structural change* and that exist only in the newer model revision, as well as the nested elements assigned with a *Move* if the new container is the current element. Examples are the number of parameters added to an operation or the number of classes that have been moved to a package.

Removed children. In contrast to inserted children, this metric counts all nested elements that have been removed from the current element. I.e. the nested elements that are assigned with a *Structural change* and that exist only in the older model revision, as well as all nested elements assigned with a *Move* if the old container was the current element. Examples are the number of parameters removed from an operation or the number of classes that have been moved out of a package.

Unchanged children. This is the number of nested elements that are not marked with any change, e.g. the number of parameters of an operation that have remained unchanged.

The values computed on nested elements can be summed up in the container element and any further ancestor element in the composition hierarchy. For example, we can calculate the number of operations inserted in a package.

Differentiation between attribute and reference changes. So far, we can count inserted, removed, changed, and unchanged elements inside a container element. In order to better differentiate the changed elements we additionally compute more fine-grained metrics that count the changes on the level of attributes and references. The set of computable metrics is given by the set of attribute types and reference types defined for the respective element type. For each attribute or reference type we can easily count the number of changes and the number of unchanged instances. If an element type, X, defines two attribute types, Y and Z, we can compute four metrics: number of elements of type X whose attribute Y has been (un-)changed, and number of elements of type X whose attribute Z has been (un-) changed.

For one model element exactly one attribute for each attribute type exists, e.g. an element has *one* name. At the same time, the model element can have many references of the same type pointing to different elements, e.g. references from UML states to outgoing transitions. However, the number of references stays usually rather low. Hence, the metrics counting attribute and reference changes are often either 1 or 0, which indicates whether there is a change of an attribute or reference of that particular type or not. Again we accumulate the computed values in the ancestor elements in the composition hierarchy.

For example, we can count how many visibilities of classes in a package have been changed, or we can count the number of transitions that point to a new target (i.e. the target reference has been changed).

9.3.2 Significance Metrics

The metrics presented so far are structural in the sense that they are solely based on the graph structure of the models (most notably containment relationships), and the structure of a difference. Metrics defined on this basis can be called *syntactical* because they do not consider the importance of changes, which depends on the semantics of a model type. For example, the following changes are both simple attribute updates: a) the change of the name of a parameter of a UML operation, b) the change of the visibility of the operation. Obviously, the latter change can have much more significant consequences. A designer who wants to get an overview of how a model has been changed from one revision to the next, is mainly interested in the significant changes.

The significance of a change depends on the semantics of a model type and cannot be deduced from metamodels. These metamodels define only the syntactical structure of models. Information about the significance of changes has to be specified separately.

```
<Nodetype name="operation"
    insert="noncritical" delete="critical" move="medium">
  <Attribute name="name" change="medium"/>
  <Attribute name="isAbstract" change="critical"/>
  <Attribute name="ownerScope" change="medium"/>
  <OrderedAttribute name="visibility"
      values="private,package,protected,public"
      increase="noncritical" decrease="critical"/>
  <Reference name="returnType" change="critical"/>
</Nodetype>
```

Listing 9.1: *Part of the specification of the significance of changes*

Specification of the significance of changes. We propose to classify changes according to the following categories: *critical*, *medium*, and *noncritical*. An example is given in Listing 9.1; we have used an XML representation of the specification here that is also used in our prototype implementation (see Section 11.1). Other representations are possible, too. We support individual classification of changes for each attribute type of each model element; the same applies to reference changes.

Metaattributes can have ordered domains, e.g. the visibility attribute has the ordered domain $\{private < protected < package < public\}$. In such cases the significance of a change can depend on the direction of the change, i.e. whether the value is increased or decreased. We propose to define the significance of attribute changes separately for the increase and decrease. An example is also given in Listing 9.1.

Similarly we can classify insertion, deletion, and move of subelements. Different classifications can be defined for each element type. For example, the insertion and deletion of operations in classes can be weighted different.

Counting changes of different significance. Given a classification we can easily declare a change to be critical, medium, or noncritical. For each model element we sum up the number of changes of a certain class. Furthermore, we differentiate between the types of changes. We produce metrics such as number of critical updates or number of noncritical insertions. We also divide by the types of model elements. Once again the values are accumulated in container elements and further ancestors. For example, we can calculate for each class in a package how many of its operations have been changed critically.

9.3.3 Similarity Metric

We assume that the differencing algorithm provides us with the similarity $sim(e_1, e_2)$ of each pair of corresponding elements. A similarity value of 0 expresses that the elements are not similar at all, the value 1 indicates identical properties. Since high values of all other metrics express dissimilarity, we transform this value into the *degree of change* (DoC) as follows:

$$DoC(n) = 1 - sim(e_1, e_2).$$

9.3.4 Aggregation of Metrics

In addition to accumulation of values by summation, it is possible to define more specific metrics in grandparent elements and their ancestors using other aggregation functions. In particular, one can compute for each element the maximum, minimum and average number of changed, inserted, deleted, and unchanged grandchildren. The elements can again be filtered by their type and other metrics can be aggregated similarly.

An example is the maximum number of changed parameters in the operations of one class. Another example is the average number of removed attributes in all classes of one package. These aggregated metrics enable a better assessment of differences.

Aggregation of the *degree of change* metric is especially interesting. The average degree of change and the maximal degree of change of direct children or grandchildren can give useful hints at the character of changes: e.g. similar average and maximum values indicate uniform changes, whereas high maximum and low average values indicate changes to specific subelements. Furthermore, the similarity of two elements, which is computed during difference computation, can be defined arbitrarily and may aggregate the similarities of all their subelements. For example, the similarity of two classes could be defined by their local properties, the similarities of their attributes, and the similarities of their operations. In contrast, the aggregated degree of change can focus on one particular element type, e.g. the average DoC of operations of a class.

9.4 Recomputation of Difference Metrics and Changes

Changes and difference metrics strongly rely on the correspondences between two model revisions. If, for instance, two classes with different names correspond, an update of the name is reported. If the classes do not correspond, two structural changes are reported: one deletion and one insertion.

Due to the fact that we allow the supplementary creation of identification links and their deletion (see Section 8.2), the correspondences between two subsequent revisions may change at any time. In such cases we have to deduce the changes and the difference metrics again. The procedures do not really differ from those presented before. The only difference is that the correspondence table is not computed by the model comparison. It is rather created based on the identification links, and the model comparison is forced to only deduce the difference.

The software metrics focus on the model elements at a particular time and are not dependent from the corresponding elements in other revisions. Thus, they do not need to be recomputed.

Chapter 10

Querying the History to Trace Elements

Once the identification links and identities have been computed, we can easily trace model elements from one revision to another. While the previous chapters (7, 8, and 9) described the computation of identification links and evolution information, we will now show how this information is used to identify model elements across evolution. Section 10.1 describes the tracing of single model elements. The tracing of model fragments is explained in Section 10.2. In Section 10.3 we finally discuss some exemplary use cases of how the tracing can be applied in practice.

10.1 Tracing an Element

> **Definition 10.1:** The task of selecting a model element (i.e. the *suspect*) in one revision (i.e. the *source* revision) and locating the corresponding model elements in other revisions (i.e. the *target* revisions) is called **tracing query**.

Depending on the model type, *one* suspect can have *several* corresponding elements in a target revision due to the ability of copying or duplicating model elements. The elements in the target revision are called *occurrences*.

If we can trace an element from one revision to another, at least one identification path going from the suspect to an occurrence exists. Such a path is the basis of the result of a tracing query. It is called a *trace*.

> **Definition 10.2:** A **trace** is the result of a tracing query. It aggregates all identification links along one identification path between the suspect and the occurrence. It expresses the evolution of the traced element and allows an assessment of the reliability of the expressed correspondence to the found occurrence.

If multiple identification paths between the suspect and the occurrence exist, we provide the user with one trace for each path. In the same manner we return

multiple traces if the suspect can be traced to multiple occurrences in one revision (i.e. if a model element has been copied so that the descendant revision contains duplicates).

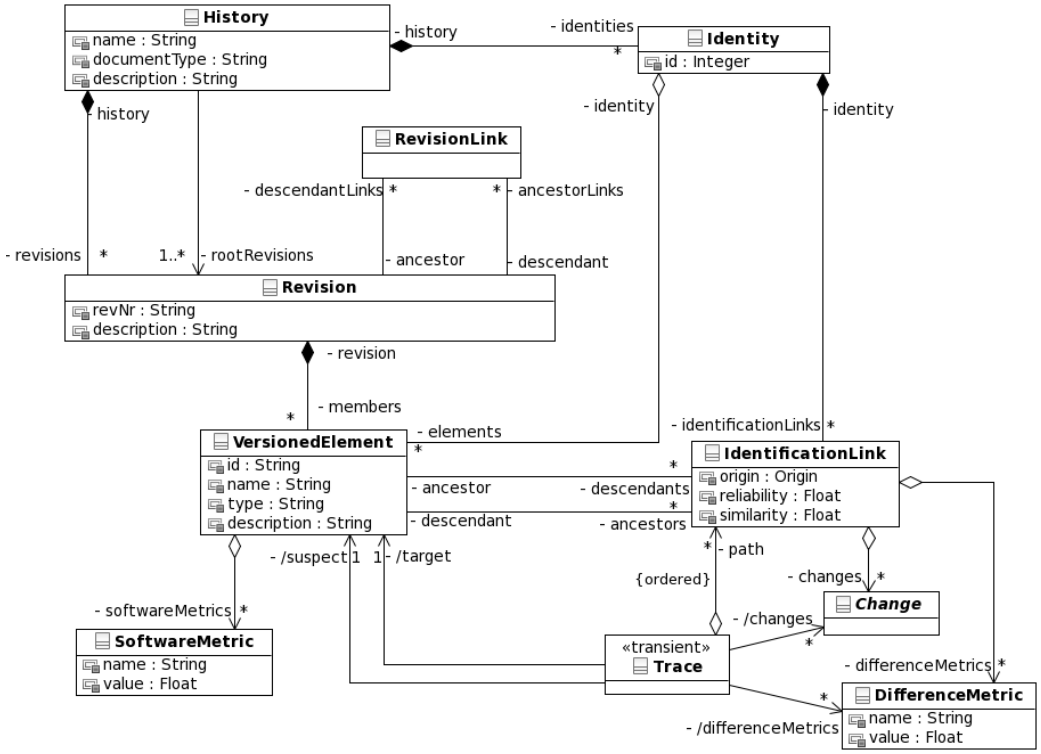


Figure 10.1: *Traces in the context of the history model*

Figure 10.1 shows how a trace relates to the information of the history. The trace connects the suspect and the occurrence (i.e. the target) and aggregates the links that build the identification path between them. The identification links are ordered from the suspect to the occurence. Furthermore, the trace refers to the changes applied to the traced elements. The difference metrics express all changes applied to the element and its nested elements along the identification path. It should be mentioned that a trace is a transient information that is not part of the history and thus not stored in the data model.

We can differentiate three types of tracing queries. They are called identification, occurrence analysis, and tracking.

> **Definition 10.3:** The **identification** is a tracing query that is executed with two input parameters: the suspect to be traced, and the target revision in which the element is to be identified. The identification returns traces that point to the corresponding elements in the target revision if existent.

If we perform an identification on all revisions of a history, we know in which revisions the particular model element exists, when it has been created, and in which revision it has been deleted. Thus, we can identify the element across its evolution. The identification can be performed with the identities stored in the history (see Section 6.3). They are sufficient to query the occurrences of a given suspect.

It is usually not sufficient to solely identify a model element in another revision, but we expect it to fulfill certain constraints.

Definition 10.4: The **occurrence analysis** is an *identification* that additionally evaluates constraints on the identified elements.
The constraints are *path-independent*, i.e. for their evaluation we only analyze the occurrences and, if necessary, the suspect; the concrete identification path is not considered.

The occurrence analysis is qualitatively on a different level than identification, since not only the existence, but even the characteristics of a model element are checked. For example, we can check if a UML association exists in another revision and if it is a composition, or we can check whether an operation exists in other revisions and if it has the same number of parameters. Occurrence analyses can be performed on all revisions of a history in order to gather, in which revisions an element fulfills the constraints. The existence is then implicitly given. An example is an analysis that queries all occurrences of a certain UML class where it has been abstract. Due to some types of constraints, the occurrence analysis may require the actual model revisions in order to evaluate the constraints; the revision objects in the history are not sufficient.

Besides pure existence we are likely interested in the evolution of a model element. Hence, we want to *track* it subsequently from one revision to another and we want to analyze the changes of the element over time.

Definition 10.5: Tracking denotes the step-wise occurrence analysis of a model element along an identification path. The constraints can therefore also consider the identification links that connect the traced element from one revision to another, or even the complete identification path between suspect and occurrence.

The tracking of a model element is particularly an analysis of subsequent revisions. It is based on the identification links stored in the history, and it is directed. Forward tracking means that we traverse along outgoing identification links, i.e. we track an element of an older revision to the newest revisions. Backward tracking follows incoming identification links and searches the oldest revisions in which

the element exists. Here we can evaluate *path-dependent* constraints on the links we traverse, and we can evaluate path-independent constraints on the versioned elements we reach over the links. If all constraints are fulfilled, we recursively traverse over the identification links of the versioned elements that we reached just before. The traversed links form a trace. It lasts from the suspect to the ultimate element that fulfills the constraints, and the last link that fulfills the constraints respectively. We say that the trace *breaks off* at the first identification link that does not fulfill the constraints or whose opposite elements do not fulfill the constraints. The lastly reached revision is then called the target revision. Examples of constraints are that the model element must not be changed along the identification path, or that the similarity should not fall below a certain threshold, or that an OCL constraint has to be fulfilled in the respective model revision.

Constraints. The constraints that can be defined for occurrence analysis or tracking depend on the purpose of the analysis. The constraints of an occurrence analysis are path-independent. We distinguish constraints that consider only the occurrences (i.e. *target constraints*) and constraints that take also the suspect or the trace into account (i.e. *comparative constraints*). The tracking can use path-independent and path-dependent constraints.

Most target constraints can be defined in OCL [111] or as a query on the software metrics; they require the occurrence to have a certain characteristic. Our approach supports the evaluation of such constraints. However, the constraints may differ entirely for different application scenarios and for different model types. As a consequence and due to its independence from any traceability aspects, we do not integrate the evaluation of target constraints into our approach, but we delegate it to the application or tool that uses our approach. Our approach provides the information about occurrence of elements in different revisions. The tools based on our approach can provide the further analysis. Examples will be shown in Sections 11.3 and 12.3.

The comparative constraints and the path-dependent constraints require traceability information and cannot by delegated to external constraint checkers. The comparative constraints basically compare the potential occurrence with the suspect. It is only reported as an occurrence if all constraints are fulfilled. The path-dependent constraints include even the identification path into the analysis.

We provide the following set of constraints that can be evaluated in both ways, comparatively and path-dependently.

Equality. The equality constraint requires the traced element to be unchanged. If this constraint is used comparatively, the suspect and the occurrence must

not differ. If it is evaluated path-dependent, even the corresponding elements in all revisions on the identification path must not differ.

Similarity. The similarity constraint checks whether the similarity value exceeds a pre-defined threshold. The threshold is a parameter of that constraint. Again, we can use it comparatively if we just consider the direct similarity between suspect and occurrence. If we use it path-dependently, each identification link on the path has to exceed the threshold.

Unchanged property. The unchanged property constraint requires that a certain property of the traced element remains unchanged. For example, the traced element is not allowed to be renamed. Hence the links on the identification path must not be assigned with a change (update, reference change, or move) that is given as a parameter of the constraint. The constraint is fulfilled in spite of a change if the constraint is comparatively used in an occurrence analysis and the change of the property has later been revoked. I.e. the identification path contains also another change of the same property that restores the original value.

If all links on an identification path originate from hash-based correspondences, the constraints “equality”, “similarity”, and “unchanged property” are always fulfilled. We further provide the following path-dependent constraints.

Difference metric. The difference metric constraint evaluates a query on the difference metrics that are assigned to the identification links on the path to be true, e.g. “Number of critical changes < 3” or “Number of inserted parameters = 0”. It considers all changes that have been applied over time. The query is given as a parameter of that constraint.

Reliability. The reliability constraint checks whether the reliability of the identification links exceeds a pre-defined threshold. The threshold is a parameter of that constraint.

10.1.1 Assessment of the Traceability

Whenever an element is traced, it is very important to assess the reliability of the result. The user must be able to appreciate the traceability information he is provided with. The information required to assess the reliability can be derived from a trace.

As shown before, each identification link has been enriched with a reliability value that expresses how trustworthy the found correspondence is. It is derived

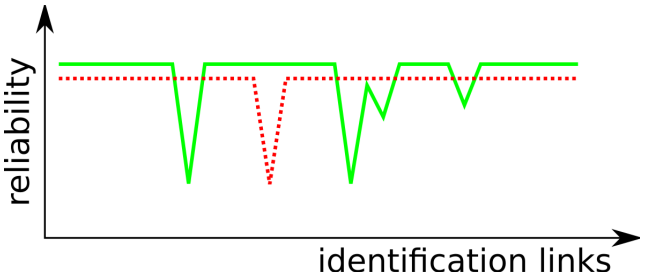


Figure 10.2: *Example of identification paths that are equal regarding average and minimum reliability*

from the algorithm that computes the correspondences. We use that information to assess the reliability of the trace.

It is not sufficient to compute one single value that expresses the overall reliability of a trace. Taking values such as the average or the minimum does not reflect the thorough reliability. An example is given in Figure 10.2 that plots the reliabilities of the identification links of two traces. Although the trace represented by the green (solid) line contains more unreliable links than the trace represented by the red (dotted) line, the averages over all reliabilities of each trace do not differ. The minimum is also the same in both cases.

Multiplication of all reliability values is also not sufficient. Since the reliability of a single link ranges from 0 to 1, the result of the multiplication is almost 0 for nearly all cases. An assessment is only possible if it includes each single reliability value on a path. As a consequence we propose to plot the reliabilities and to allow the assessment on a visual basis. The user can quickly capture how reliable a trace is. He can furthermore see the identification links that are suspicious and subject them to a more detailed review. Obviously, the visual assessment of the reliability of a trace can only be utilized in semi-automatic applications so that a user interprets the visualization. For fully automated scenarios, we recommend to use statistical methods for the interpretation of the reliabilities.

The similarity of two elements connected by an identification link is another indicator that enables an assessment of the reliability. Seen over time the similarity gives a first impression of how frequently the traced element has been changed, and a high similarity underpins the probability that the found element is the correct element. We propose to visualize the similarities together with the reliability values. Further indicators that can be considered are the distance between the suspect and the occurrence (i.e. the number of identification links in between), and the origins of the identification links.

10.1.2 Assessment of the Evolution

As said before, we are not only interested in the pure identification or tracking of an element, but also in its evolution. We are most likely interested in the difference between the suspect and the occurrences. In addition it is reasonable to comprehend *how* the element has evolved, i.e. the changes that have been applied in the intermediate revisions.

The difference can easily be deduced by aggregating the changes that are assigned to the identification links of a trace. Therefore we have to eliminate overlapping changes; if two changes focus on the same property, we ignore the older change, i.e. the one that has been applied before the other change.

The evolution can be captured by the aggregation of the difference metrics that we have computed for each identification link. An advantage of the metrics is that they even include the changes that have been applied to the nested elements of the traced elements.

Aggregation over time. If the length of a trace (i.e. the distance between the source revision and the target revision) is large, there are many intermediate revisions and each may contain several changes. We can aggregate the difference metrics along the trace, so that they compactly express all changes that have been applied to the traced element.

It is absurd to confront the user with each single change that has been applied to the model elements; the changes let the user drown in a plethora of information and he will not be able to get an overview and to capture the relevant changes. Instead of listing each single change, we can better aggregate the difference metrics along the trace.

An example is a short history of a UML class model; it contains a class called “book” that was renamed to “booklet”, later to “manuscript” and finally to “document”. If the user selects this class in the first revision as suspect and the last revision is the target revision, the user is not confronted with each single renaming, but he only sees that the class “book” is now called “document” and that it has been renamed three times. The old and the new name are obviously important to know; intermediate names can be neglected. The information that the class has been renamed more than once, however, informs the user about the high dynamics in the elements’ evolution. Considered together with the number of intermediate revisions, it indicates a probability for further evolution. If we assume only three or four intermediate revisions in our example, we could assume that the class will again be renamed in the next revision that will be created.

Aggregation over the neighborhood. Besides assessing the changes applied to the traced element over time, we are often interested in the context of the element, i.e. its neighborhood, as we can rarely see a model element decoupled from its context. An example is a state of a UML state machine. Although the state might be unchanged regarding its local properties compared to the ancestor revision, it might be that the transitions do now connect this state with other states than before. In this case the state probably has other semantics despite being locally unchanged. Sometimes the context is not only given by the direct neighborhood, but more distant elements or even the whole model has to be considered as the context.

A listing of each single change of the neighborhood or the whole model might become very large and cannot be overviewed due to the large amount of elements that might have changed. It is again profitable to use difference metrics, which can be aggregated. The user can capture the amount of change applied to the context. Due to the significance metrics, he is further able to point out important changes more quickly.

It is furthermore possible to aggregate the metrics of the neighborhood over time. For each intermediate revision we can take the aggregated values that express the differences of the neighborhood, and we can then aggregate these values over time. Hence we are able to assess the evolution of the traced elements and their neighborhood or even the whole model with a concise set of numbers.

10.2 Tracing Model Fragments

Very often the user is not interested in tracing single model elements over time, but model fragments. Informally we can say that a model fragment is a subset of a model.[1] More precisely, it is a arbitrary set of model elements that *can* stand in some relation to each other.

The user is able to select not only a single element as suspect, but a set of elements. These elements are called *members*. It is not necessary, that the members stand in a certain relation, such as they have to be neighbored. The user can rather select an arbitrary set of elements, whose interrelations are probably not obvious at a first glance. The relationships between the selected elements are analyzed by our tracing approach in order to understand the elements as a fragment.

The occurrence analysis of a model fragment is actually an occurrence analysis of each member of a fragment. If we have identified all members in the target

[1] Note that here the term *model fragment* has a different meaning than in the UML specification, which uses the term to denote different files that contain parts of a model.

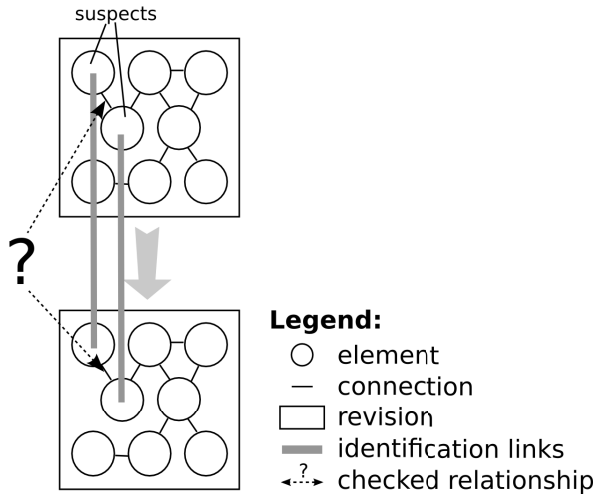


Figure 10.3: *Example of tracing a model fragment*

revision, we check whether the relationships captured beforehand also exist in the target revision. One could say that it is an occurrence analysis of multiple elements and the constraint evaluated by the occurrence analysis is that the elements bear the same relations as in the source revision. Figure 10.3 depicts an example. In the upper revision two elements have been selected. Each element is traced separately by following its identification link to the lower revision. Then it is checked whether the relationship (i.e. the edge in the internal graph representation) that existed between the members of the fragment in the source revision (denoted by the small line connecting them) can also be found in the target revision.

It should be mentioned that it is not possible to trace the relationships directly. The tracing approach deals with model elements represented by vertices in our graph representation of the model (see Section 5.1). The elements are usually visible to the user as graphical shapes in the diagram representation of the model. In contrast, the relationships between model elements are represented as edges in our graph representation. They are not first class citizens, because the user is usually not able to capture them as discrete objects. They are rather logical, as they can be encoded by attribute values in the external model representation (i.e. an attribute that contains the local identifier of another model element).

The tracking of a model fragment is performed similarly. We follow the identification links of each member and check if the previously analyzed relationships also exist between the newly reached elements. Hence, the tracking of model fragments requires the fragments to exist in all intermediate revisions between the source revision and the target revision. The trace breaks off if the relationships between the members disappear.

10.2.1 Selection of the Fragment to be Traced

Since the user is not able to select relationships explicitly, he just selects the elements that are to be traced together. This selection is carried out by defining a set of suspects (i.e. the members) in the source revision. Subsequently, the set is analyzed for direct relationships and transitive relationships between the members.

Explicit selection of contained elements. If we trace a model element that is a container for other model elements, we are not necessarily interested in tracing the contained elements. An implicit inclusion of contained elements prohibits the tracking or the occurrence analysis of the container if any contained element cannot be traced. Hence we do not include nested elements in a fragment by default. We rather allow, for example, to trace a class regardless of its operations.

Nested elements that are to be traced together with their container have to explicitly be selected as suspects. The containment relationship is detected by the analysis procedure presented next. An example is the snippet of a UML model depicted in Figure 10.4. The user explicitly selects the class "A" and its attribute "b"; the containment relationship between "A" and "b" is selected by our analysis of direct relationships (denoted by the symbol R_d).

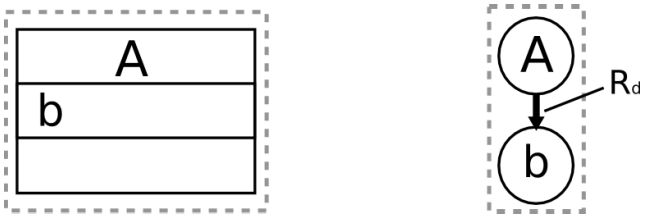


Figure 10.4: *Example of the selection of containment relationships*

Analysis of direct relationships. In order to reveal direct relationships we check for each element v of the set of members M whether it refers to another element w of the set, i.e. there is a edge pointing from v to w. We can define a set R_d that contains all edges representing such relationships between the members of the traced fragment.

$$R_d = \{e \in E \mid src(e) = v \in M \wedge tgt(e) = w \in M \wedge v \neq w\},$$

where E is the set of edges in the graph representation, and M is a subset of the vertices in the graph ($M \subseteq V$).

Hence, the fragment now consists of the elements selected by the user *and* the direct relationships R_d revealed by our analysis. The elements are no longer a plain set, but a fragment of the model. However, it is still possible that the set of members contains model elements that do not share any relationship.

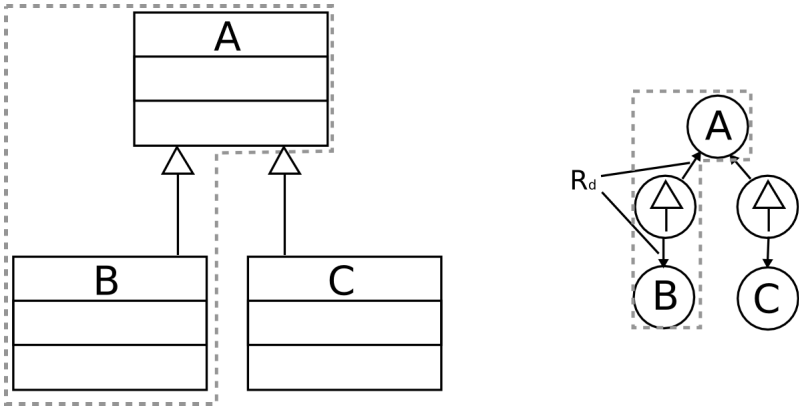


Figure 10.5: *Example of the selection of connected model elements*

Figure 10.5 shows a snippet of a UML model (left) and the internal graph representation (right). In this example, the user has selected the classes "A" and "B" and the generalization between them (dashed rectangle). Although the user is not able to explicitly select the relationships between the generalization and the classes "A" and "B", it is part of the internal selection as it is contained in R_d.

Analysis of transitive relationships. Often a fragment is not just defined by a set of model elements with direct relationships in between. An example are UML classes that are connected by associations. We can see two different levels of relationships.

1. The model elements are not connected directly, but with an extra *connection element* in between. For example, we assume that the UML metamodel does not contain a direct relationship between the association and its source and target classes, but elements of type association end express the relationships.[2] Figure 10.6 depicts this situation. The user can select the classes "A" and "C" and the association "b" in the external representation of the model (at top of the figure). The internal representation (bottom) also shows the association ends, which are illustrated by the vertex with the diamond and the vertex with the arrow; they need to be included in the model fragment.

[2]Actually, the UML metamodel defines properties that correspond to our association end elements, however, they are either contained by the association or the classes depending on the navigability.

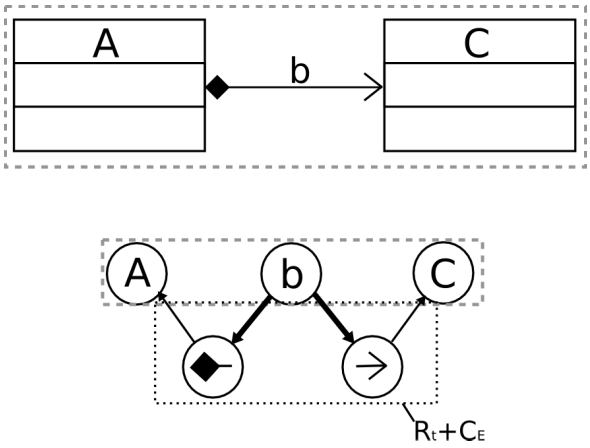


Figure 10.6: *Example of the selection of a model fragment*

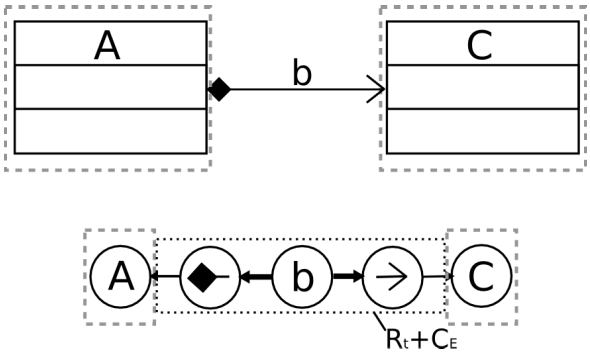


Figure 10.7: *Example of another selection of a model fragment*

2. Model elements can also stand in relation to each other if they are connected by a set or a path of other model elements. An example is shown in Figure 10.7. Here the user depicts only the classes, however, the association expresses a relationship between the classes. Hence, the association and the association ends could be included in the identification.

We can see that it is not sufficient to consider only direct relationships, but the elements forming a relationship need to be revealed, too. We create another set of relationships R_t. We therefore use the ability to query paths on our internal graph representation. For each element v of the set of members M we check whether a path p according to a query $q \in T_P$ exists that points to another element w of the set. We populate R_t with the edges of the path and we define a set C_E that contains the elements included in that path except its source and target, which have already been selected as members (i.e. the connection elements).

$R_t = \{e \in E \mid e \in p \wedge src_p(p) = v \in M \wedge tgt_p(p) = w \in M \wedge query(p) = q \in T_P\}$ and

$$C_E = \{x \in V \mid x \in vertices_p(p) \ \wedge \ x \neq src_p(p) \ \wedge \ x \neq tgt_p(p) \\ \wedge \ src_p(p) = v \in M \ \wedge \ tgt_p(p) = w \in M \ \wedge \ query(p) = q \in T_P\} \ ,$$

where E is the set of edges and V is the set of vertices in the graph representation, M is a subset of the vertices in the graph ($M \subseteq V$), and T_P is the set of path types (queries) that are to be evaluated on the members. The edges of direct and transitive relationships can be unified in the set R_M that represents all edges contained in a fragment:

$$R_M = R_d \cup R_t$$

For the example in Figure 10.7, we have defined a path expression that starts from a class. It goes over the incoming target reference to an association end and from there over the container reference to the association. From there it navigates further over the nesting reference to the other association end. It finally follows the target reference to another class (i.e. the class connected by the association). The different types of paths T_P are defined separately for each type of model element and metamodel. Arbitrary many types of paths can be defined for a model element type. A class in UML, for instance, has further paths that point to other classes, e.g. the path over a generalization edge or the path over the type references of the classes' attributes. We recommend that the user selects the paths that are queried in order to reveal the relationships, since the user is probably not interested in all possible relationships. The paths are usually queried with the function $eval_C$ that does not reveal paths that contain cycles (see Section 5.3). We can also define queries to be evaluated with the function $eval$. They allow us to select elements that are neighbored to a fragment, e.g. if the user selects a generalization element, we can automatically select the super class and the subclass connected by the generalization. We should mention, that the querying of paths can only be performed on the graph representation of a model; it cannot be performed on the versioned elements in the history, since they do not reflect the structural relationships.

Figure 10.8 illustrates a data structure that can be used to represent the model fragment to be traced. A *fragment* object represents the selected fragment. It acts as container for all information that is required to define the fragment. The aggregation *members* refers to the members of the fragment. The direct relationships between the members are represented by objects of type *DirectRelationship*. The transitive relationships are represented by objects of type *TransitiveRelationship*, which in turn refer to the connection elements and the relationships between them. It should be mentioned that this is a temporary data structure; it is not part of the history.

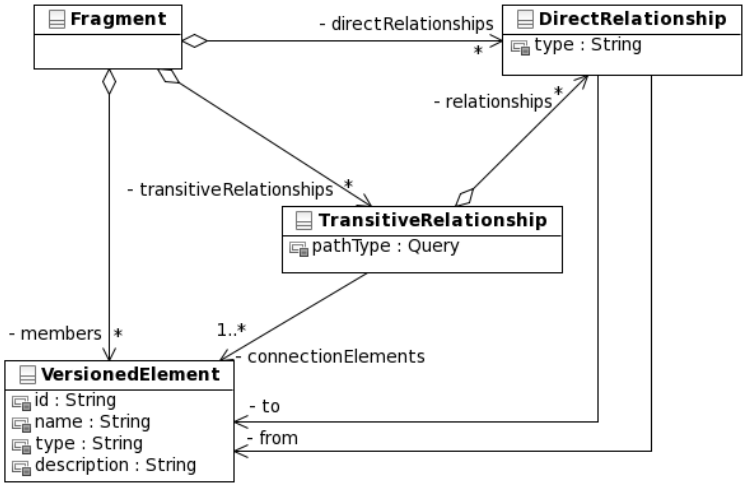


Figure 10.8: *Representation of a model fragment*

10.2.2 Checking the Existence of Fragments

As described before, we first trace the members of a fragment and then we check whether the relationships revealed from the source revision also exist in the target revision. Hence the existence of the traced fragment in another revision requires that all members can be identified in the target revision. If this condition is not fulfilled and the fragment does not exist in the target revision, an occurrence analysis would report no occurrence, and the tracking would stop.

If all members can be traced successfully, we check whether the elements still form the same model fragment, i.e. the relationships between the elements are still existent. In the same manner as we have computed the sets of relationships in the source revision (i.e. R_M and C_E) we can compute the respective sets of relationships in the target revision, R_M^t and C_E^t. The connection elements of C_E can also be traced individually. Their identification in the target revision is the second requirement for the model fragment to be traced, i.e. an identification path (cf. Definition 6.6 on page 68) in between exists.

$$\forall\ v \in C_E : \exists\ v' \in C_E^t : \exists\ ip(v, v')$$

Finally we have to check whether each relationship in R_M has a corresponding relationship in R_M^t:

$$\begin{aligned} \forall\ e \in R_M : \exists\ c' \in\ & R_M^t \mid src(c) \text{ can be traced to } src(c') \\ & \text{and } \ tgt(c) \text{ can be traced to } tgt(c')\ . \end{aligned}$$

If all of these requirements are fulfilled, we have successfully traced the model fragment to another revision.

Depending on the use case it is not always required that all relationships of a model fragment exist in the target revision. The user is usually interested in the elements corresponding to the member of the fragment and the additional information whether these elements still form the same fragment as in the source revision due to their relationships. The single relationships and the connection elements play a subordinate role for the user. They only become interesting for the assessment of the tracing. We further allow that some of the members are optional, i.e. they are not required in order to trace the fragment, e.g. if we trace a UML class with all its subclasses, we are also interested in the occurrence of the classes if the inheritance is missing, i.e. the generalization elements can be set to be optional.

Checking further constraints. For the occurrence analysis and the tracking of single model elements we have introduced the ability to check constraints on the found elements. The same mechanism is provided for traced model fragments. As for single elements, we support the evaluation of comparative and path-dependent constraints. The evaluation of path-independent constraints should be performed by the tool that uses the tracing approach. Especially for model fragments the additional constraints can be arbitrarily complex and use case specific.

10.2.3 Assessment of the Tracing

The tracing of a model fragment returns a set of traces, so that each member and each connection element of the fragment is assigned with a separate trace. If we performed an occurrence analysis, we would get a second instance of the fragment data structure as presented before (see Figure 10.8). This fragment represents the occurrence of the fragment in the target revision. If we tracked the fragment, we would get an instance of that structure for each revision along the traces. The traces and the fragment instances are aggregated in a fragment trace. Figure 10.9 depicts the respective data structure.

In order to assess the found occurrences of a traced model fragment we should first assess the completeness of the occurrences. Hence we calculate the ratio between found and traced members, and the respective ratios for direct and transitive relationships.

The reliability can again be derived from the reliabilities of the single identification links of the traces. Since we now trace multiple elements at a time, it is even less sufficient to compute a single value expressing the reliability. Again, we propose an assessment based on a visualization. If we plot the reliability values of all traces, the visualization becomes very crowded and we cannot capture the

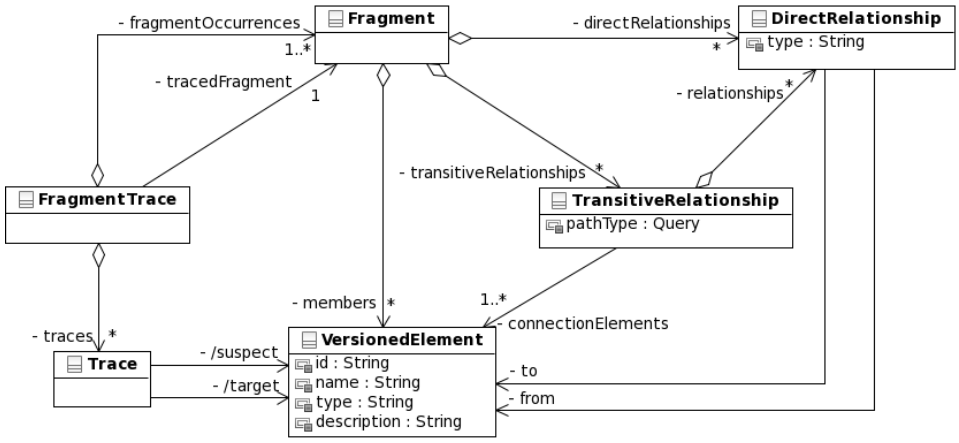


Figure 10.9: *Structure of a fragment trace*

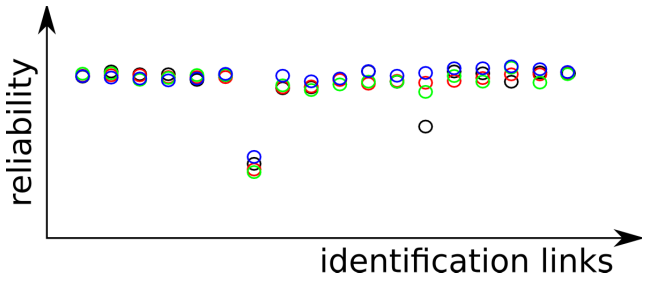


Figure 10.10: *Example of a plot of all reliability values of a traced fragment*

reliabilities of each single member we traced. However, it gives an overview of the reliabilities of all members at a time. An example is given in Figure 10.10. Already, for four traces the plot does not enable the reliability assessment of each member, but we can see that all traces have a low reliability at the same step. It is recommended to review that particular step in more detail.

While the plot of all reliabilities enables an overall assessment of the traces of the fragment, we support the analysis of the reliabilities of each member by the means of box plots. Box plots are easy to comprehend. Each box illustrates the reliability of the identification links of one trace. The boxes show the range of the reliability values. The box vertically starts at the 1st quartile and ends at the third quartile. The median is shown by a bold line inside the box. The whiskers show minimum and maximum values if they stay within 1.5 standard derivations above or below the median. Values outside this range are called outliers and marked by small circles. We can display several box plots at a time, so that the user can get an overview of the reliabilities inside a trace. Figure 10.11 gives examples. The trace of element “A” has a very good reliability. The majority of the identification

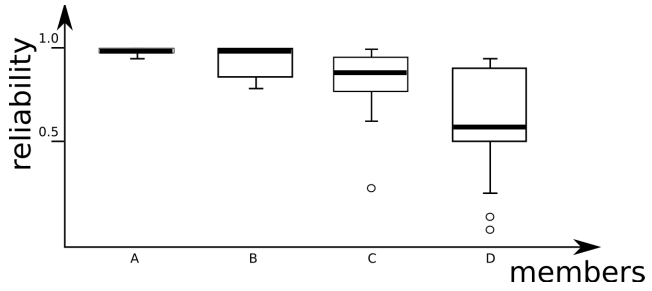


Figure 10.11: *Examples of the box plots showing the reliabilities of traces of members*

links of the trace have a reliability of 1. The exceptions have still high reliabilities. Hence we can trust in the trace of element “A”. The trace of element “B” has more identification links with a lower reliability, however, they are still very good, so that we can trust in that trace, too. For element “C” an identification link with a rather low reliability exists. We should inspect that trace in more detail. This can be done by plotting the reliabilities of that single trace in the same manner as for tracing single elements. Element “D” in our example can only be traced over several identification links with low reliability values. We better should not trust in that trace. A manual inspection is required.

Again, the similarities can be regarded for the assessment of the traces. We recommend to visualize the similarities as box plots, too. Hence we get pairs of box plots for each member of the traced model fragment. The interpretation of the box plots remains the same.

The evolution of the traced model fragment can be captured by aggregating the difference metrics that are assigned to the identification links of the traces of the members. Since the metrics express the differences in numbers, we can easily accumulate the values of the metrics of all members, so that we can assess the evolution.

10.3 Application Scenarios

In order to get an understanding of the different tracing tasks we briefly want to introduce typical use cases in which the tracing is applied. Therefore we pick up the scenarios and questions we have described in Section 1.5.1 and Section 1.5.2.

10.3.1 Typical Scenarios

External Links: A very regular use case for tracing model elements is the linking by external tools, i.e. an external tool points to the elements inside a model revision. An example is a requirement engineering tool that assigns each model element to one or more requirements. The link between requirements and model elements are mostly realized in a way that the local identifiers of the model elements are stored within the requirements engineering tool. In order to recover the links for a newer revision of the model, it is mostly sufficient to perform an identification. However, the evolution data computed by our approach is very important information, since the user is enabled to check whether the linked model element has been changed that extensively that it cannot any longer be assigned with the requirement.

External links also occur if several models are linked to each other so that elements may refer to elements from other models. A model that contains references to elements in another model is from the technical point of view the same as a tool. However, in order to weave linked models, it is often necessary that linked model elements have to fulfill certain constraints, e.g. classes are linked if they have a common super class. The linking itself can then be assigned with OCL constraints or the like. If newer revisions of the models are to be weaved, we can perform an occurrence analysis that checks that the additional constraints are fulfilled. If they are not fulfilled anymore, the weaving can be prohibited in order to avoid incorrect models.

Evolution Analysis: Evolution analysis deals with the inspection of a model over time. Usually it measures certain metrics for the model or for certain elements inside the model. For example, we want to measure how the model or certain elements have grown over time. For this purpose we recommend tracking. It usually starts at the creation of the inspected element and ends with its deletion regardless the number of changes applied meanwhile, because the user is most probably interested in the complete lifetime of the suspect. Thus, constraints do not have to be defined. A lot of evolution information can already be queried from the traces that are returned by the tracking. Information about changes is given by the change objects and difference metrics. Furthermore, the software metrics that are stored in the history can be queried for each revision in which the element(s) exist. They can be analyzed as part of the evolution analysis that uses our approach. It is not necessary to limit the evolution analysis to the data stored in the history. It is also possible to query only the identification information and to run separate analyzes such as logical coupling analysis.

Metamodel Evolution: If a metamodel evolves, it is often wanted to migrate the existing instances of that metamodel to the newer version. Therefore, it is necessary to uniquely identify the metaobjects of the old version in the newer one. If the newer metaobjects are identified, one can create transformation rules to automatically migrate the instances. The rules usually have to be created with a lot of expert knowledge. If they are created one after another, identification can be used to assist the developer in finding that metaobject in the newer metamodel that corresponds to the metaobject of the older metamodel. However, the task is still very tedious. Sometimes an occurrence analysis can be performed with additional constraints that a metaobject has to fulfill.

Model Merging: In collaborative and distributed development the different variants of a model are often developed in parallel. If variants are managed as different branches in the software configuration management system, we can support the merging of models. Occurrence analysis can be used to check whether the elements of one variant also exist in the other variant or in the base revision. A simple three-way-merging approach compares the variants to each other and to the base revision. In contrast, occurrence analysis considers also the revisions between the base revision and the variants. Hence, it is more precise and ensures that the merged elements have the same identity. Furthermore, the definition of constraints can be used to prohibit incorrect merging. The evolution information assigned to the traces can be used for the analysis of merge conflicts.

Bug Tracing: A use case that strongly utilizes constraints in an occurrence analysis is bug tracing. In this case, we have a suspect or a set of suspects that are involved in a bug, and we want to locate occurrences of that bug in other revisions. We assume that the bug is only present in another revision (and should be fixed there) if the set of elements involved in the bug occurs as a whole and with only very small changes in the other revision. Therefore, similarities of the elements in different revisions must exceed an additional threshold and the number of critical and medium changes has to be zero to consider the elements as a repetition of the bug.

10.3.2 Answering Typical Questions

Besides using traceability information for the scenarios presented above, it can be used to answer even very concrete questions. Subsequently, we list different questions and describe how they can be answered.

- How old is the given element? / In which revision has the given element been deleted?
  Track the element backwards / forward.
- In which revisions and/or variants of the model does the given element(s) exist?
  For an element: Query the identity of the versioned element and check all other versioned elements that are contained in that identity.
  For a set of elements: Create a fragment and perform an occurrence analysis on all revisions in which you are interested.
- How much/often has the given element been changed from a certain point in the past until now?
  Track the element and perform a query of the change objects or the difference metrics that are assigned to the returned trace.
- Are the given element of the given model revision and the equally named element in another model revision the same?
  Check if both versioned elements are assigned to the same identity.
- When was the given element renamed the last time?
  Track the element backwards with a constraint that prohibits an update on the attribute "name" or a difference metric constraint saying "Number of changed attributes of type 'name' = 0".
- Since when does the given group of elements exists? Did it appear in one step or has it "grown"?
  Create a fragment with all elements being optional members. Then track the fragment backwards.
- Have the elements of the group changed their connections among each other?
  Trace the fragment and analyze the relationships and connection elements in the found occurrence.
- Did elements of the given fragment disappear or are they just no longer connected?
  Create a fragment so that the probably unconnected elements are optional members. Then perform an occurrence analysis in the respective revisions.

We see that the history can be easily queried to answer a broad spectrum of questions. However, the list of questions, as well as the application scenarios we discussed previously, are not limited to those mentioned here. They should rather give an impression of the possibilities.

Part IV

Evaluation

Chapter 11

Prototype Implementation

For a proof of concept, we have implemented our tracing approach in a prototype. We developed a tracing service that can be integrated into different applications. It includes the history representation, the computation of identification links and evolution information, and functions to trace model elements and fragments over time. The implementation of the service is described in Section 11.1. The role of the underlying differencing approach SiDiff and the necessary modifications to it are discussed in Section 11.2. In addition we present the prototype of an interactive tracing tool with a graphical user interface in Section 11.3. It is based on the implementation of our tracing service and verifies its applicability.

11.1 Implementation of the Tracing Service

We have implemented our tracing approach as a service based on the OSGi platform [118]. The OSGi platform is a framework that allows the realization of component-based software systems. Each component (i.e. called a *bundle* in OSGi) can provide several services. A service is basically a plain Java object that can be used to fulfill a certain task. The OSGi platform provides a central registry and routines for registration, binding, and execution of services. OSGi is widely accepted for Java systems and builds the basis for the integrated development environment Eclipse [35]. Thus, the implementation of our approach is ready to use in Eclipse and Eclipse-based applications.

The usage of OSGi allows us to smoothly integrate the SiDiff engine into our approach, because it has also been realized on basis of the OSGi platform (see Section 4.3). The kernel of SiDiff is thereby divided into several components that provide different services each. This enables a sufficient reusability, because a single bundle can be reused at different locations in the realized software. We can thus reuse a lot of functionality that is offered by SiDiff, e.g. for evaluating expressions or reading XML-based configuration files.

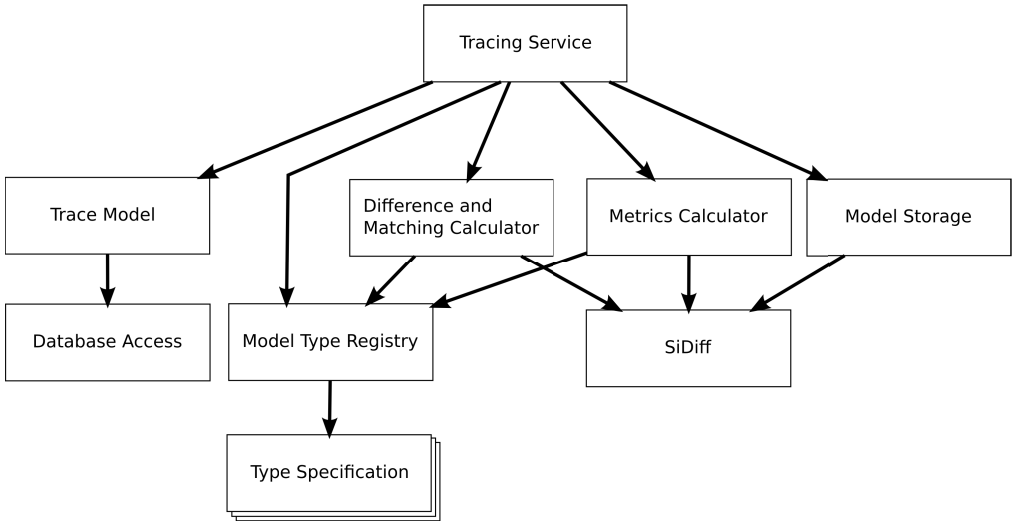


Figure 11.1: *The architecture of the tracing service*

It is a common design pattern in OSGi-based applications to pack the definition of a service and its implementation in separate bundles. This provides a high flexibility, because components have only references to the interface bundles that define the services. The implementation can be exchanged anytime by replacing the implementation bundles with others. We are thus able to realize our implementation of the tracing approach on the basis of the interface bundles of SiDiff without being dependent from the concrete implementation.

11.1.1 Architectural Overview

Figure 11.1 illustrates the architecture of our tracing service. The main component *tracing service* contains the core routines of the tracing approach, such as traversing the revisions of a history, computing identification links, etc. It also defines the OSGi service *TracingService* that acts as an interface of our approach.

The tracing service uses the history data model as presented in Chapter 6. The model is an object-oriented interface to all traceability and evolution information that is computed by our approach. Due to the large amount of data, we have implemented the model as an object-relational wrapper (i.e. a transient data structure), so that all information is stored in a relational database. The access to the database is encapsulated in the *database access* component in order to keep our implementation independent from particular database management systems. For our prototype we have used the MySQL database system [117].

The model revisions are managed in a component called *model storage*. While the history data model stores only references to revisions and versioned elements

(i.e. in the database), the model revisions are given by the original serialization files of the respective modeling tool (i.e. in the file system). The model storage manages the files in the file system and maps the revision objects of the history onto them. It handles the deserialization and transformation of model revisions into the internal graph representation. It further provides a cache to keep the revisions to be compared in the memory, so that we do not have to reload them if a revision is compared to an older ancestor that has been analyzed earlier. The model storage is partly based on SiDiff, which already comes with routines for loading, transforming, and managing models. However, SiDiff is not required from the conceptual point of view, but it was reused for convenience.

In order to keep our implementation independent from particular metamodels, we realized a component called *model type registry*. This component manages the different types of models that are supported by our tracing service. Each model type is described by a *type specification* component. The type specification holds the metamodel, the transformation rules to map models onto our internal graph representation, and all configuration parameters in order to run our tracing approach on a history of models of that type. Examples of configuration parameters are the configuration files for SiDiff and the coefficients for the reliability calculation.

The *difference and matching calculator* encapsulates the computation of correspondences and differences with SiDiff. It configures and executes SiDiff according to the configuration parameters of the type specification for the current type of model. We are thus able to compute this information for different types of models. It is based on the interfaces of the different services and bundles of SiDiff, so that the implementation can be exchanged.

The *metrics calculator* computes the difference metrics used for assessing the evolution of traced elements. It is also based on the SiDiff kernel, although other difference engines could also be used. The metrics computation is also parameterized by a type specification so that differences of model revisions of arbitrary types can be measured. Furthermore, the metrics calculator can compute a broad number of software metrics that can be realized by counting operations on the model revisions. We realized the metrics calculator as a separate component, because we think that especially the computation of difference metrics can also be applied to other use cases that are independent from our tracing approach.

11.1.2 Model Representation with EMF

The internal graph representation of model revisions that keeps our approach independent from particular types of models has been realized with the Eclipse

Modeling Framework (EMF) [31]. EMF is compatible with the internal graph representation presented before. The *EObjects* in EMF represent elements and are thus the vertices of our graph; the edges of our graph are given by the references between EObjects. The types of vertices and edges are given by *EClass* and *EReference* objects in EMF. The attributes of an element type are defined by the *EAttribute*s. All information inside a model revision can be accessed through the reflective interface functions given by the EMF framework.

In addition, the Eclipse Modeling Framework allows us to define the graph schema explicitly. It is basically the metamodel and can be extended with metainformation, such as type-specific configuration data, and we can technically ensure that all revisions of a history conform to the same metamodel.

With EMF we can represent models of arbitrary type. EMF realizations of different metamodels already exist, e.g. the UML2 implementation [36], and new metamodels can easily be created with EMF. Many modeling tools use EMF as basis for their internal model representation, and they use EMF serialization schemas to store their models in files. Hence, with the usage of EMF we ease the application of our tracing approach in existing modeling environments. Another advantage is that the current version of the SiDiff engine natively supports EMF-based models. We can apply the SiDiff comparison to the histories, which we analyze, without expensive transformation steps.

However, we even support external model representations that are not based on the Eclipse Modeling Framework. If the serialized model revisions are given in an XML format we can transform them with XSLT [151] into an internal representation. In case of non-XML formats, such as the *.mdl format of MATLAB/-Simulink™, we have to implement a parser that reads the proprietary files. Due to the fact that SiDiff uses the same format, however, we are able to reuse all transformation and parser components of SiDiff.

We cache the internal graph representations in the file system by means of the (de-)serialization mechanism of EMF. The original model revisions have to be transformed into the internal graph representations just once. Whenever the graph representation is needed again, e.g. if constraints are evaluated during an occurrence analysis, it can be loaded from the file-system without performing the transformation again.

11.1.3 The History Data Model

All traceability and evolution information can be accessed through the history that we presented in Chapter 6. In general, the data model has been implemented as defined before. Figure 11.2 depicts the design. It has been extended by conve-

nience operations for better access to the stored information.

The *History* contains an operation *getRevision* that returns the revision identified by the revision number given as a parameter. It is not necessary to traverse of the whole history in order to get a particular revision. Similarly, the revision contains an operation *getElement* that returns the versioned element representing the model element with the given identifier. The operation translates between elements of the external representation of the model revision and the respective versioned elements if the same identifiers are used.

The *Identity* has been enriched by the operation *getElements* that returns all versioned elements assigned to the identity in a particular revision that is given as a parameter. This allows us to easily identify an element of one revision in another. The Java statement would be:

```
suspect.getIdentity().getElements(targetRevision),
```

where *suspect* is the versioned element representing the element to be traced and *targetRevision* represents the revision in which we want to identify the corresponding element of the suspect.

The *IdentificationLink* has got three additional operations. The operation *getDistance* returns the distance between the ancestor and the descendant of the link. Usually it is 1, but for gaps it provides information about the gap size. The operation *getChanges* returns a list of changes that have been applied to the element along that link. A single difference metric can be queried with the operation *getDifferenceMetricValue*.

The *VersionedElement* has also been enriched by a convenience operation, *getSoftwareMetricValue*, that allows us to query single metric values.

The *Trace* has got many additional operations. The distance between the suspect and the target can be queried with the *getDistance* operation. *getChanges* aggregates all changes that have been applied to the traced element along the trace. A single difference metric can be queried with the operation *getDifferenceMetricValue*; the metric aggregates all changes along the trace. The operations *getAllReliabilities* and *getAllSimilarities* return map objects that store for each revision the reliability and the similarity that are assigned to the link pointing to that revision. Furthermore, we inserted operations for querying the minimum and the average reliability and similarity of all identification links of a trace. The operation *getDirectSimilarity* performs a direct comparison of the suspect and the occurrence and returns their similarity. With *getRatioOfOrigins* we can query the ratio of identification links with the given type of origin to all identification links of the trace. Finally, the trace object provides access to the metrics of the traced elements. Software metrics and difference metrics can be queried as maps that

store for each revision the value of the given software metric and the difference metric that is assigned to the link pointing to that revision.

Due to the limitation of the heap space in Java virtual machines and the large size of history information, we have implemented the history data model as an object-relational wrapper. It allows us to load only the relevant parts of a history into the memory. We always create an object representation of the history and the revisions, because there is only one history object and the number of revisions is also rather low. The number of Java objects representing versioned elements, identification links and identities, however, can explode with the size of the model and the number of revisions. As a consequence, we only create Java objects if needed. Each time a versioned element, identification link or identity is requested, a new object is returned to avoid cross references that contradict Java's garbage collection. *Trace* objects are not persisted in the database. They are rather temporary objects that are created if model elements are traced. This implementation enables a scalability in size of models and in the number of revisions a history can have.

We encapsulated the data behind a set of interfaces that only provide reading access to the stored information. From outside the history can be used to query traceability and evolution information, but it neither allows the creation or manipulation of histories nor the computation of identification links and evolution data. This functionality is accessible through a service interface.

11.1.4 The Service Interface

The tracing approach can be used through a single interface given by the class *TracingService* as depicted in Figure 11.3. It provides operations to create and extend histories, to access existing histories from the database, and to trace elements or fragments.

New histories can be created with the *createHistory* operation. It is parameterized with the name, the type, and a description of the history. The name is used to identify the history within the service as it supports the management for arbitrary many histories. The type defines the type of the model whose history is to be analyzed. The type information is checked whenever a new revision is added to the history, so that we do not compare apples with oranges. The description is for describing the history; it can be left empty. All values are used to create an object of the type *History* of our data model. This object is returned and can be used to access traceability and evolution information later.

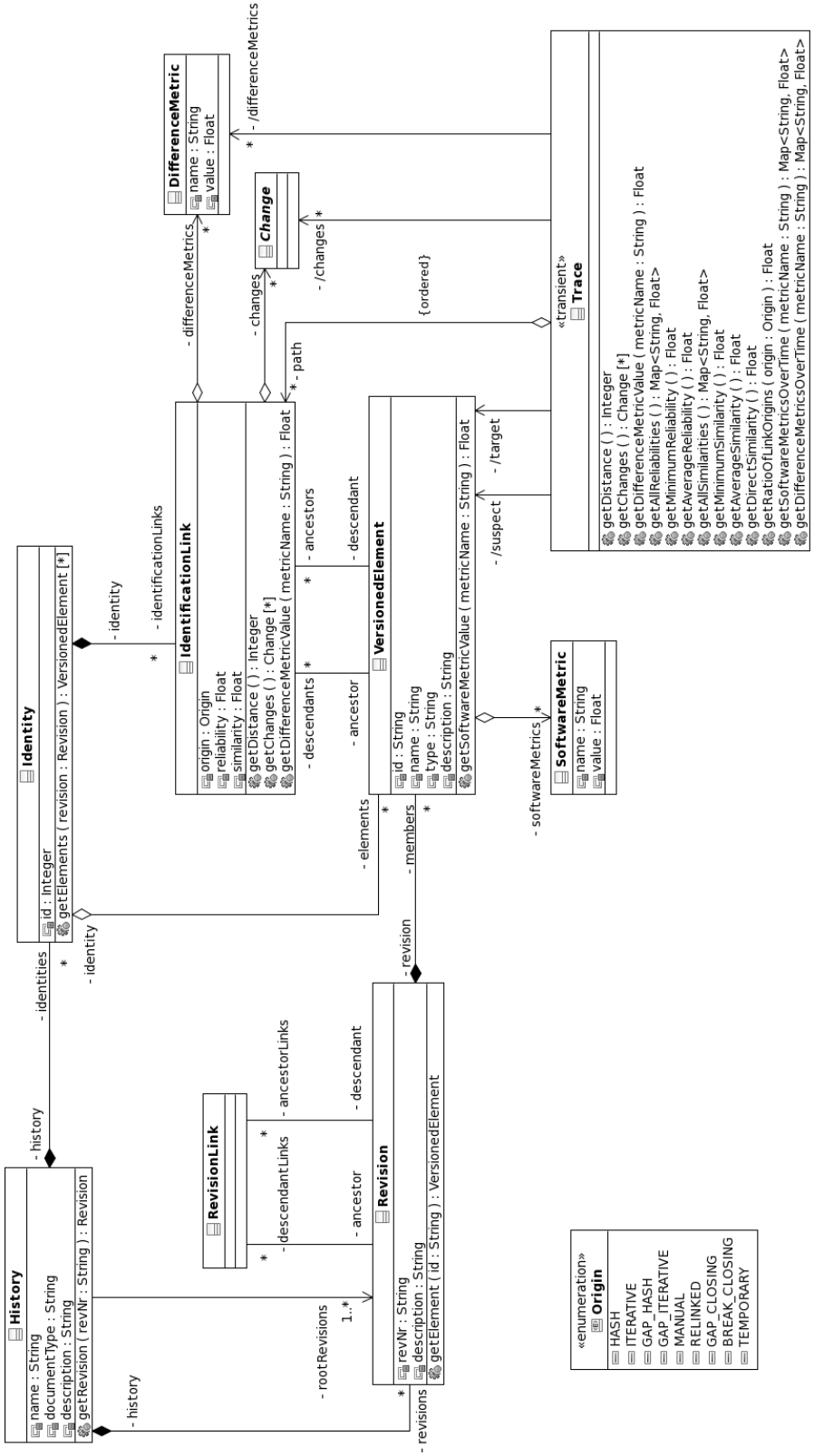


Figure 11.2: *The implemented data model for histories*

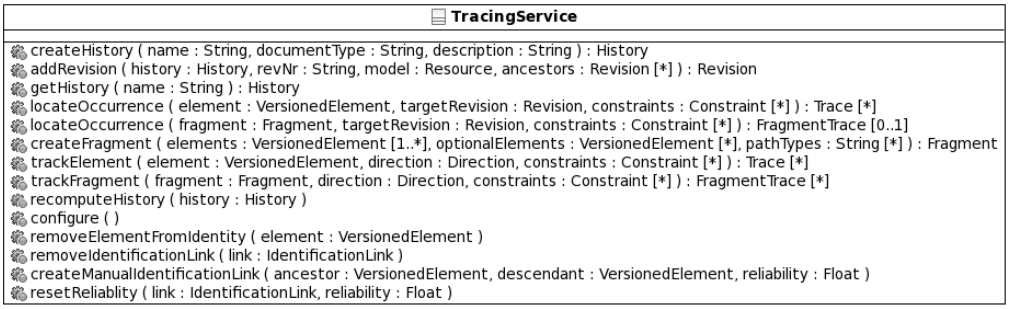


Figure 11.3: *Interface of the tracing service*

Existing histories can be requested by their name. It is given to the *getHistory* operation, and the respective history object is returned. This allows to access the tracing information from histories that have been analyzed earlier.

Once a history has been created or loaded from the tracing service, we can add new revisions to that history by calling the *addRevision* operation. It requires the history, a revision number and the resource as parameters. Optionally, one or more ancestor revisions can be given. The revision number is the unique identifier of the revision within the history. The resource is an EMF representation of the model revision. The operation returns an object of type *Revision* of our history data model. It represents the revision and can be used as parameter for further calls of the *addRevision* operation in order to declare this revision to be the ancestor of another revision. If no ancestor revisions are given as parameter, the added revision is a root revision of the history. Whenever a new revision is added to the history, all traceability and evolution information is computed as described in Chapter 7. This requires that revisions are added in their chronological order. We do not provide a function for inserting a revision in the middle of the history.

Interface for tracing model elements. The tracing service provides two operations to trace model elements. An occurrence analysis can be performed with the operation *locateOccurrence*. It requires a suspect element, a target revision, and an optional set of constraints as parameters. If the suspect can be traced to the target revision, the operation returns the respective traces otherwise it returns an empty set.[1] The tracking of an element can be triggered with the *trackElement* operation. It requires the element to be tracked, a direction (i.e. forward or backward), and an optional set of constraints that have to be fulfilled while tracking the element.

[1] Our approach supports that an element can be traced to multiple elements inside one revision if the element has been copied. However, the current implementation of the underlying SiDiff approach does not report such cases.

The operation also returns trace objects for each occurrence in the target revision (i.e. the farest revision to which the element has been tracked).

In order to trace model fragments, the tracing service offers the operation *createFragment* that enables the definition of a fragment. It is called with a set of members and a set of optional members. It furthermore requires a set of path types in order to reveal the transitive relationships between the elements. The fragment definition is returned as an object of the type *Fragment*. It contains the members as well as their relationships. The tracing can be triggered with the operations *locateOccurrence* and *trackFragment*. They have to be used in the same manner as the operations for single model elements, however, they expect a fragment instead of a versioned element, and they return objects of the type *FragmentTrace*.

Figure 11.4 shows the design of fragments and fragment traces. It implements the data structure presented in Section 10.2. *Fragment* acts as container for all information that is required to define the fragment. We differentiate between members and optional members, which both are versioned elements. We represent the direct relationships by objects of type *DirectRelationship*. The transitive relationships are represented by objects of type *TransitiveRelationship*, which in turn refer to the connection elements and the relationships between them. Furthermore, the fragment offers operations that return the numbers of the different types of objects. Occurrences of the traced fragment in other revisions are represented by *FragmentOccurence* objects. In contrast to the former model in Section 10.2 (Figure 10.9), we have now realized them as subclasses of the fragment. The class *FragmentOccurence* provides an operation to compute the ratio of members in the found fragment to the number of members in the traced fragment (*getMemberCompleteness*). Similar operations are provided to compute the ratio of found relationships. The operation *getTraceOfMember* returns the trace of a particular member of a found fragment. All occurrences of a traced fragment are aggregated in a *FragmentTrace* object. It is returned by the trace operation *locateOccurrence* and *trackFragment* of the tracing service. It provides an operation to query the occurrences of the traced fragments in a particular revision.

Configuration and re-computation. The operation *recomputeHistory* can be used to recompute all traceability and evolution information of the given history. Thus, the user is able to reconfigure the tracing approach, e.g. by manipulating the comparison configuration of SiDiff. All information can then be recomputed without recreating the history, i.e. we do not have to add all model revisions again. The *recomputeHistory* operation can also be used to compute the information at a stretch if we switch off the behavior to analyze revisions immediately when they

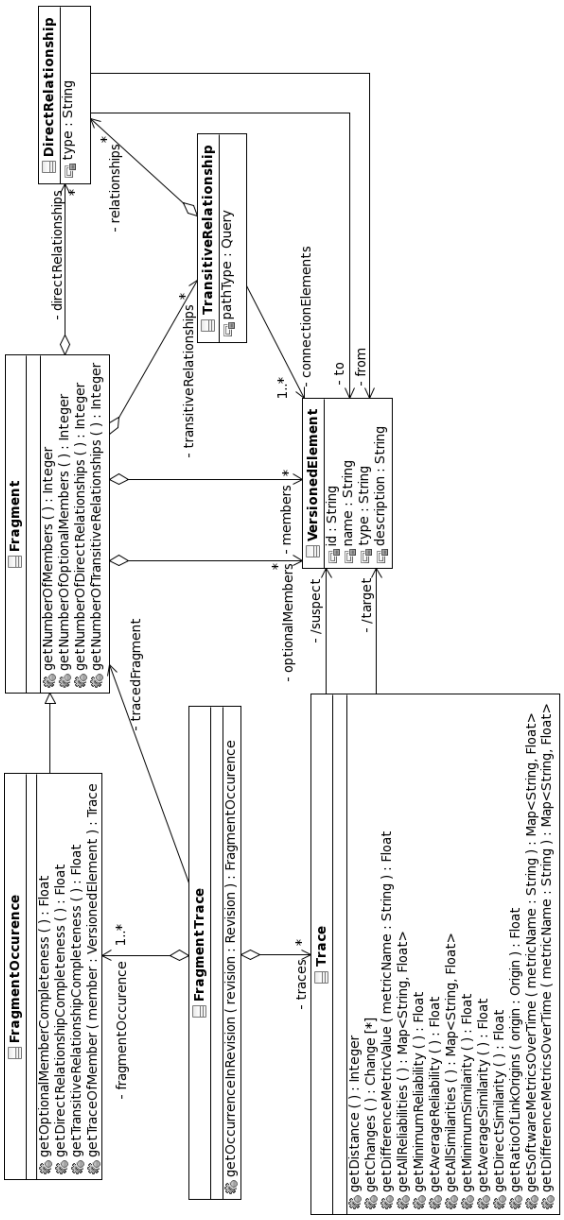


Figure 11.4: *The design of fragments and fragment traces*

are added to the history.

Furthermore, the tracing service provides several configuration operations that allow us to set global parameters for the tracing approach, indicated by the *configure* operation in Figure 11.3.[2] We can set all information needed to connect to the

[2]For each parameter exists a get operation and a set operation, however, in Figure 11.3 we indicated them by one single operation in order to save space.

relational database that stores the history (i.e. IP and port of the database server, database name, user and password). We can further define technical properties, namely the folder where the original model files are stored in the file system and the size of the cache for holding the model revisions to be compared in memory. And we can configure the functioning of our implementation by setting the maximum size of gaps (i.e. to how many ancestors revisions a revision is compared) and whether tracing information is computed immediately when a new revision is added or only on explicit request.

Manipulation of traceability information. In order to manipulate the traceability information computed by our approach, the tracing service provides one operation for each correction task as described in Section 8.2. The operations have not been realized as operations of the classes of the history data model to strictly separate between reading and editing of traceability information. The operation *removeElementFromIdentity* removes the versioned element given as a parameter from its identity. The element is automatically assigned with a new identity. A single identification link can be removed with the operation *removeIdentificationLink*; it takes the link to be removed as a parameter. New links can be created by calling the *createManualIdentificationLink* operation. Besides the elements to be linked, it requires the reliability value that expresses the belief of the user. With *resetReliability* the user can assign a new reliability value to an identification link.

11.1.5 Computation of Difference Metrics

The computation of difference metrics is independent from any traceability scenario, hence we have realized it as a self-contained module. It requires a tree-like graph representation of the model revisions, a correspondence table that denotes pairs of model elements that correspond, and a difference computation engine that provides us with the following types of changes:

- changed attributes,
- changed references,
- structural changes (elements that exist only in one revision), and
- moved elements (i.e. the reference to the container has changed)

Optionally, the correspondence table may contain information about similarities between each corresponding pair of elements. However, from the point of view of difference metrics the similarity is just a numerical value between 0 and 1; its computation is concern of the differencing engine.

In our case we already have a tree-like graph representation of the analyzed model revisions (see Section 5.1). The correspondence table is filled with the information given by identification links; they provide the correspondences and similarity values. We use SiDiff as difference computation engine, since it is already used by the tracing approach. Thus, no additional engine would be required.

11.2 Usage of the SiDiff Toolbox

The core problem that was to be solved in our tracing approach is the computation of correspondences between the elements of different model revisions. We utilized the differencing tool SiDiff to reveal the matching. In order to decouple the implementation of our approach from SiDiff, we encapsulated the SiDiff algorithm behind a facade class. The facade provides four methods:

compare() runs a normal comparison of two model revisions. It computes the similarities, reveals correspondences, and computes the difference. It is used when two subsequent revisions are compared (see Listing 7.1).

compareDistantRevisions() is similar to *compare()*, however, it narrows the computation down to a given set of model elements. Hence, SiDiff does not compare the revisions completely, but only some parts. Since this method is used to compare two revisions that do not stand in direct ancestor-descendant relationship, the correspondences between the elements that are not compared are set according to the existing identification paths in between (see Listing 7.3 and Listing 11.1).

computeSimilarities() computes the similarities between all elements of two revisions that do not stand in ancestor-descendant relationship. The correspondences are therefore set according to the identification paths connecting the elements, and the similarity computation is performed. This method is used to compute the similarities if gaps are closed (Listing 7.4) and if the *getDirectSimilarity()* operation of a trace is called (see Section 11.1.3).

computeDifferences() is similar to *computeSimilarities()*, however, it computes the difference. It is used for the recomputation of differences and difference metrics whenever identification links are manually created or deleted.

Each method can be parameterized to write the computed data into the database before it is returned to the caller. If the method is called for the same revisions again, the data from the database can be returned.

As exchange format for the computed data, such as correspondence information or similarity tables, we used the data structures that are provided to us by SiDiff.

Hence we can work with the data without converting it beforehand. However, our implementation does not refer to any concrete implementation but only refers to the interfaces defined by SiDiff. All SiDiff services except some general utilities (e.g. for filtering sets, managing files, etc.) are loosely coupled by the configuration of the OSGi platform that defines which concrete implementations are bound to the interfaces.

11.2.1 Modification and Extension of SiDiff

Besides the use of existing services of SiDiff we had to apply some modifications and extensions. The *compareDistantRevisions()* operation performs a comparison of just a part of the model. In order to do so, we configure SiDiff to use a special candidates service, and we create an initial matching based on the identities before we run the SiDiff algorithm. Listing 11.1 shows the modifications. SiDiff uses a candidates service to check which elements of two revisions are candidates for each other, i.e. whether they can correspond or not. While the standard candidates service of SiDiff just checks for the elements to be matched if they are of the same type and if they are not matched yet, the newly implemented candidates service (i.e. the *TracingCandidatesService*) enables the explicit setting of candidate pairs.

We create an instance of this service (line 3) and create pairs of candidates so that each candidate element of the first revision can be matched only with an equally-typed candidate element of the second revision (lines 4–10). SiDiff is configured to use this candidates service (line 11), so that it can only produce correspondences between the elements of the beforehand computed candidate sets.

As shown in Section 4.3.2, the SiDiff algorithm does not only analyze local properties to find correspondences, but it also uses information about neighbored elements (e.g. if they have been matched already). In order to reveal correspondences between the candidate elements, we have to inject the correspondence information about the elements that we have traced already. We iterate over all elements of the first revision that are not in the candidate set (line 13). If the element can be traced to the second revision, we create a correspondence (lines 14–17). After the correspondences have been created, we can call SiDiff and return the result (line 18). Due to our special candidates service, SiDiff can only reveal correspondences between the candidate elements of the compared revisions (i.e. the elements which are not linked yet). The similarity computation within SiDiff, however, can work without any modification as the information about correspondences between neighbored elements has been injected as initial matching.

```
1  function compareDistantRevisions(Revision r1, Revision r2,
2                                   Set c1, Set c2) {
3    CandidatesService ca = new TracingCandidatesService();
4    for each Element e1 in c1 {
5      for each Element e2 in c2 {
6        if (e1.getType() == e2.getType()) {
7          ca.addCandidatePair(e1,e2);
8        }
9      }
10   }
11   SiDiff.setCandidatesService(ca);
12   CorrespondencesService co = SiDiff.getCorrespondencesService();
13   for each Element e1 in r1\c1 {
14     Element e2 = e1.getIdentity().getElements(r2);
15     if (e2 != null) {
16       co.addCorrespondence(e1,e2);
17     }
18   }
19   return SiDiff.compare(r1,r2);
20 }
```

Listing 11.1: *Modified SiDiff variant to compare parts of a model*

The *computeSimilarities()* operation computes the similarities between two models. Usually, the similarities are computed by the SiDiff algorithm, which iteratively computes the similarities and the correspondences as long as new correspondences can be found. The similarities are thereby distributed over the whole model similar to the similarity flooding algorithm (see Section 4.3.3). Here, we set the correspondences directly and run the similarities calculator afterwards. We repeat the similarity computation three times, which is the average number of iterations of the unmodified SiDiff algorithm. Therefore, we enriched SiDiff with an external interface to the similarity calculator and to the similarities table.

While our approach deals with objects of the types *Revision* and *VersionedElement*, SiDiff requires EMF *Resource* objects that contain the model elements as objects of type *EObject*. Therefore, we realized the above-mentioned operations *compare()*, *compareDistantRevisions()*, *computeSimilarities()*, and *computeDifference()* in a way that they perform an implicit translation of all objects.

Reliability computation. In order to compute reliability values for found correspondences we implemented a new service called *ReliabilityCalculator* that calculates the values as described in Section 8.1. Since we wanted to keep the changes

to SiDiff components as small as possible and we did not want to manipulate the SiDiff algorithm directly, we realized the reliability calculator as a passive component that observes the SiDiff algorithm. Therefore, we extended the SiDiff components that they fire events whenever

1. the similarities computation of a pair of elements starts or ends,
2. a new comparison rule is evaluated,
3. an existing correspondence was used by a comparison rule, and
4. a new correspondence is found.

With the first three types of events we are able to capture the similarities that depend on other correspondences in order to compute the *cleaned similarity* (see Section 8.1.3). The fourth type of event allows us to capture the state of the comparison if a correspondence is found. We can query the similarities, the other candidates, the number of elements that are still unmatched, and so on.[3] In order to store the reliability information with the respective correspondence, we adapted the correspondence service of SiDiff. The adaptation allows us to attach any additional information to a correspondence.

The coefficients that are used while calculating the reliability values are stored in an XML-based file as depicted in Listing 11.2.

```
<Class name="Class"
        minimalHashReliability="0.8"    // base
        equalPathWeight="0.1"           // a1 for hash-based correspondences
        equalParentWeight="0.05"        // a2  "
        hashContextWeight="0.02"        // a3  "
        probabilityWeight="0.1"         // a1 for iterative correspondences
        similarityDistanceWeight="0.5" // a2  "
        cleanedSimilarityWeight="0.4"  // a3  "
        parentReliabilityWeight="0.0"  // a4  "
        similarityThreshold="0.5"       // the threshold used by SiDiff
        moveAllowed="true"              // the element is allowed to be moved
/>
```

Listing 11.2: *Example of a setting of reliability coefficients*

Listing 11.3 depicts an example of the specification of the context of model elements. It is used while computing the reliability of hash-based correspondences.

[3]The reliability of a hash-based correspondence can be computed by analyzing all correspondences found by the hash-based matcher at once. It is not necessary to monitor the process of computing the hash-based correspondences.

```
<Class name="Class">
  <Context name="HashReliabilityContext">
    <Path expr="generalizations/generalElement" />
    <PathC expr="assocEnds/association/assocEnds/target" />
  </Context>
</Class>
```

Listing 11.3: *Example of a context definition*

Here, we see the context of classes in UML. It consists of the super classes and the classes referred to by associations. The element `<Path>` denotes the evaluation of path expression on the model. `<PathC>` evaluates the query without cycling (i.e. vertices are not visited twice).

Persistence of manual corrections. The manual correction of traceability information is often a tedious task. The user has to manually inspect the links in order to prove their correctness. In case of newly created links the task is even more complex, since the user has to manually reveal the correspondence information that was not found by the heuristics. As a consequence the decisions made by the user have to be made persistent. Of course, the changes applied to the history are stored, but if the identification links will be recomputed, the user decisions would get lost.

We serialize the decisions of the user. We store created links as tuples of two versioned elements and a reliability value. Removed links are stored as tuples of two versioned elements. The tuples are stored in XML files so that one file exists for each pair of revisions. Listing 11.4 depicts a snippet of such a file.

Whenever we compare two revisions during the recomputation of the traceability information, we load the file that contains the manual decisions for that pair of revisions. Before we perform the model comparison with SiDiff, we create a correspondence for each *Correspondence* defined in the file. The similarity of the corresponding elements it currently set to 1, as it is done for hash-based matches. For *NotACandidate* tuples in the file we remove the denoted elements from the list of candidates.

11.2.2 Compatibility to Other Model Comparison Approaches

The correspondence computation needed by our tracing approach can also be performed by other algorithms or tools than SiDiff. However, there is a set of requirements that have to be fulfilled by alternative approaches:

```
<ManualCorrespondenceDecisions>
  <Correspondence idA="_s034b17" idB="_r083c98" rel="1.0" />
  <Correspondence idA="_s172b87" idB="_r293c01" rel="0.9" />
  <NotACandidate idA="_s683b24" idB="_r527c63" />
</ManualCorrespondenceDecisions>
```

Listing 11.4: *Example of persisted user decisions*

- The computation has to be based on similarities, so that even elements without persistent identifiers can be matched if they have changed.
- The alternative approach should be generic or configurable to support different types of models. Otherwise we would have to adapt different approaches for different types of models.
- The correspondences and similarities computed by the approach have to be accessible.
- We must be able to set the correspondences in order to only compute similarities.
- We must be able to narrow the computation down to parts of the models.

From a technical point of view, we would have to implement adapters for the data structures of the alternative approach, so that the data is compatible to the interfaces of SiDiff that we refer to in our implementation of the tracing approach.

11.3 Implementation of a Tracing Tool

In addition to the tracing service that can be embedded in various modeling tools or environments, we have implemented a generic tool that allows us to directly trace model elements within a given history of a model. The tool has been implemented based on the Eclipse Rich Client Platform (RCP) [34]. The history view has been realized with the Graphical Editing Framework (GEF) [32], which is a framework for creating graphical editors and views. The revisions view and the details view have been realized with the Standard Widget Toolkit (SWT) [37] for creating graphical user interfaces. The tool does not compute any data itself, but it uses the tracing service that we have presented before.

Figure 11.5 shows a screenshot of the tool. The screen of the tool can be divided into three areas: a history view on the left side, one or more revision views on the upper right side, and a details view on the lower right side.

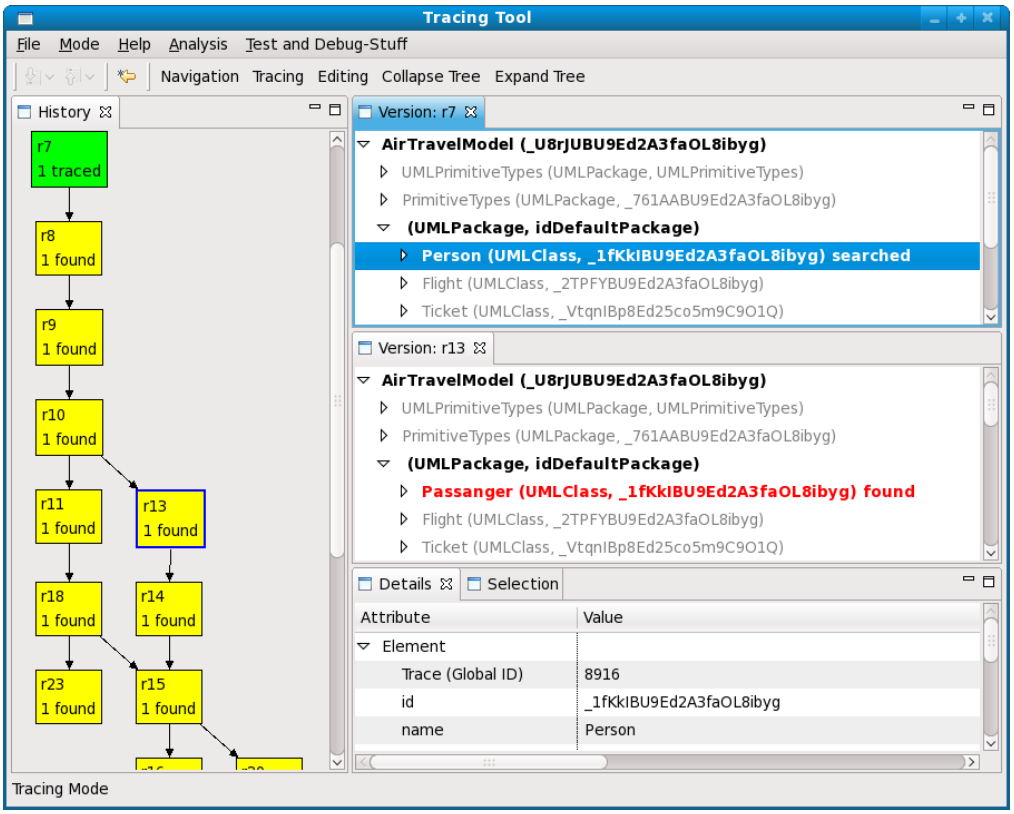


Figure 11.5: *Screenshot of the tracing tool*

The history view shows the history as a graph. Each revision is represented by a rectangle. The rectangles are connected by arrows that point from ancestor revisions to descendant revisions. The view enables the selection of revisions. If a revision is selected, details about that revision are displayed in the details view. Furthermore, if elements are traced, the details view can provide tracing information for each revision, and the revisions in the history view can be colored and labeled individually (e.g. if the traced elements occur in the particular revision).

The user can select revisions to be opened. They are then shown in a revision view on the upper right side. The revision view shows a textual tree representation of the model revision. Model elements are represented by the value of their name attribute (if existent), their type, and their local identifier. Model elements that are contained by other elements are indented below their container. It is possible to open multiple revisions at a time. Their content is shown in separate revision views that are shown as different tabs, which can be arranged side-by-side. The revision view allows the user to inspect a revision of the model. Elements can be selected, and information about the selected elements is shown in the details

view. It provides a list of key-value pairs of the elements' attributes, and it displays traceability information such as the corresponding elements in the ancestor revision and in the descendant revision.

Besides information about the currently selected revision or elements of that revision, the details view is used to provide information if the user traces single elements or model fragments.

Tracing. In the revision view the user can select the elements to be traced. The ancestors and descendants are always shown in the details view. It is also possible to directly navigate to them. A context menu enables the user to perform an occurrence analysis and a tracking of the selected elements. The occurrence analysis is performed on the complete history, i.e. for all revisions. The user can define the constraints to be checked in an extra dialog.

If a single element was selected, the tool queries the tracing service. All revisions in which the traced element can be identified are highlighted in the history view. The user can immediately overview the revisions in which the element occurs. If a highlighted revision is opened in the revision view, the corresponding elements are marked, and the details view shows further information, such as the reliability of the identification, the similarity between the suspect and the found elements, and the changes that have been applied. Furthermore, we provide a visualization of the reliability and similarity values of the trace in a line chart.

If more than one element was selected, we treat the selected elements as a model fragment. The tracing service is used to create an object representation of the fragment that is subsequently traced. The user has to define which of the selected elements are optional in the fragment and he can select the relationships to be evaluated in an extra dialog. In another dialog the user can define the constraints to be checked. Again the occurrences are highlighted in the history view. Here, we also denote, to which extend the fragment has been found. If a revision is opened, the found members are again highlighted and further tracing information is shown in the details view. The reliability and similarity values of the traces can be visualized as box plots as proposed in Section 10.2.3.

Since the occurrence analysis has short execution runtime, we do not require that the user defines a target revision explicitly. We rather highlight all revisions in which the element or the model fragment can be identified. The user can immediately see whether the actual target revision contains a correspondence. In addition, the coloring of the history view reveals the lifetime of the suspect. The user can see since when the element or the fragment exists (i.e. the revision in which it occurred the first time), and when it has been deleted. Revisions in which the suspect was temporarily removed (i.e. gaps) can immediately be revealed, too.

Analysis functions. The tool also provides functions to analyze the tracing information. It can highlight all versioned elements of a history that are assigned to a single identity (i.e. the elements exist only in one revision). The revisions that contain such untraceable elements are colored in the history view and the number of elements is denoted in the revisions label. If revisions are opened in the revision view, the untraceable elements are highlighted. The user can immediately locate the untraceable elements. Similarly, the tool can highlight the beginnings and the ends of identities, i.e. the elements that miss an identification link to either an ancestor or a descendant.

If an element misses an identification link, the tool can analyze whether the ancestor revision, and the descendant revision respectively, contain an element with equal identifier, equal name, or equal path, that also misses an identification link. These elements should be further inspected if they could correspond.

In addition, the tool can be used to locate the elements whose identification links have an insufficient reliability value. Therefore, the user can define a threshold and all elements that have an identification link whose reliability value does not exceed the threshold are highlighted.

In order to review suspicious elements (i.e. elements that are untraceable, miss an identification link, or have an unreliable identification link), the tool can display the difference between two revisions. Therefore, it triggers the difference computation and presents the changes in a list and a parallel representation of the compared revisions. The representation is technically based on the user interface of the Eclipse plugin EMFCompare [30].

Manipulation. For the case that the review of suspicious elements reveals errors in the traces, the tool provides functions for manipulating the traceability information of elements according to Section 8.2. After switching into an editing mode, the user is provided with context menus to remove elements from an identity, to remove single identification links, and to create new identification links. All tasks are delegated to the corresponding operation in the tracing service.

Chapter 12

Case Studies

Based on the prototype implementation of our approach (see Chapter 11) we performed different case studies in order to evaluate our tracing approach. In one empirical study we performed controlled experiments with the histories of different models. The experiments and their results are described in Section 12.1. In another empirical study with an earlier prototype of our approach we let test persons assess the applicability of our approach to real analysis scenarios. The study is described in Section 12.2. In Section 12.3 we briefly introduce other analysis tools in which our approach has successfully been used.

12.1 Validation of the Approach

We have performed controlled experiments to validate the traceability information computed by our approach. We subsequently describe the experiments and the data we use. Afterwards, in Section 12.1.2 we discuss the results of the experiments.

12.1.1 Study Design

The best way to validate the traceability information computed by our approach, namely the identification links, is comparing them with alternative traceability information. However, an alternative approach to compute traceability information does not exist. Thus, alternative traceability information must either be captured manually or persistent identifiers have to be used. We were not able to get test data that has manually captured traceability information, but we have test data that is either enriched by persistent identifiers or identifiers can be generated from the data.

Hence, we validate the identification links computed by our approach against the information given by persistent identifiers or identifiers generated from the

properties of model elements. Due to the fact that persistent or generated identifiers are not very reliable (see Section 2.1.1), our validation also includes a manual inspection of the results. Whenever the information computed by our approach differs from the correspondences implied by the identifiers, we manually check which information is correct.

Test procedure. For each model we create a history with the tracing service presented in Section 11.1. All revisions are subsequently added to the history and the traceability information is computed. Afterwards, we compare the identification links computed by our approach to the traceability information implied by the persistent or generated identifiers.

We perform a precision-recall analysis which is often used to evaluate the quality of information retrieval approaches [10]. The analysis compares the sets of relevant and irrelevant data with the data that has been reported by an information retrieval algorithm. *Precision* measures the exactness of the results; *recall* measures completeness. Since the computation of identification links can also be classified as an information retrieval problem, this test is adequate to evaluate our approach.

We perform the precision-recall analysis for each pair of subsequent revisions of a history separately. First, we compute the sets of relevant and irrelevant data. We check for each element of a revision, whether its identifier also exists in the ancestor revision. If the identifier was found in the ancestor revision, we assume that the element also exists in the ancestor revision. Otherwise we assume, that the element is new. Elements that exist in the ancestor revision should be found by our approach, i.e. they should be connected by an identification link. Hence, we assign them to the set of *relevant* data. New elements, however, should not be connected by an identification link, and are thus assigned to the set of *irrelevant* data.

In the next step, we analyze the data retrieved by our approach, namely, the computed identification links. According to the precision-recall terminology, the elements that have an identification link that points to the ancestor revision are called *positives*. Elements without an identification link pointing to the ancestor revision are called *negatives*.

We check whether the computed data is correct and classify the positives and negatives to be either correct (i.e. called *true* in precision-recall terminology) or incorrect (i.e. called *false*). If the link points to an element that has an equal identifier, the element is called a *true positive* (i.e. it is in the set of relevant data and has been found). If the link points to an element that has a different identifier, we have to inspect the data manually. We differentiate between two cases.

The element is not linked to an ancestor and no element with the same identifier exist in the ancestor revision.	⇒	true negative
The element is linked to an ancestor and the ancestor has the same identifier.	⇒	true positive
The element is linked to an ancestor but the ancestor represents a different model element.	⇒	false positive
The element is not linked to an ancestor but the element existed already in the ancestor revision.	⇒	false negative

Table 12.1: *Classification of the computed tracing data*

(1) The linked elements are indeed different elements and do not correspond. The link computed by our approach is thus incorrect and we mark it to be a *false positive*. (2) Although the elements have different identifiers, it can be that they are the same element at different times. In this case, the information implied by the identifiers is incorrect. We reassign the element from the set of irrelevant data to the set of relevant data and we mark the identification link as a *true positive*. Elements without incoming identification links are called *true negatives* if the ancestor revision does not contain any element with equal identifier. If such an element exists but it was not found by our approach, we have to inspect the data manually again. If the information implied by the identifier is correct, we mark the identification link to be a *false negative*. However, if the identifier is misleading, we mark the identification link to be a *true negative* and element is reassigned from the set of relevant data to the set of irrelevant data. Table 12.1 summarizes the classification of the computed tracing data.

Test data. We analyze different types of models, namely UML class models created with the Rational Software Architect [60], Ecore models created with the Ecore editor of the Eclipse Modeling Framework [31], and Simulink models created with MATLAB/Simulink™ [136].

The UML class models have identifiers assigned to the model elements. The identifiers are persistent and can be used as alternative tracing data, because the Rational Software Architect retains them during deserialization and serialization.

For the Ecore models we are able to generate identifiers from the properties of model elements. Therefore, we take the value of the name attribute of the elements and compute a qualified name by concatenating the names of all container elements. E.g. the operation *doSomething* of class *Bar* in package *foo* is assigned with the identifier *foo/Bar/doSomething*. If a model element is without name at-

No.	Type	No. of Revisions	No. of Elements (min/max/avg)
A	UML class model	19	2/138/99
B	UML class model	10	103/130/118
C	Ecore model	31	96/178/129
D	Ecore model	73	8017/11828/8776
E	Simulink model	4	6500/6803/6617

Table 12.2: *Histories used for controlled experiments*

tribute, we compute the name with an individual rule, e.g. for a generalization, we take the names of the connected classes. The generated identifiers can be used as alternative tracing data. However, we have to review each identifier change manually, because the surrogates might change if the model is changed.

MATLAB/Simulink™ creates identifiers on the basis of element names similar to the identifiers we generate for Ecore models, hence, an alternative traceability information for Simulink models is given.

Table 12.2 gives an overview of the models we have used for the controlled experiments. It lists the models, their type, the size of their history (in number of revisions), and their minimal, maximal, and average size per revision (in number of elements).

History A contains a UML model that describes different entities of an airport. It has particularly been created for demos and for testing purposes. It provokes some special cases such as gaps in the traces. The model given by History B is the data model of an evolution analysis tool. The UML model describes basically the database schema that is used to represent the data stored in repositories, namely files, revisions, check-in comments, etc. History C contains an Ecore description of one schema available in SiDiff to serialize difference information. History D is a copy of the UML2 project hosted on the CVS server of Eclipse. It contains the Ecore implementation of the UML metamodel [36]. Four snapshots of a Simulink model of a speed control have been used to create History E. They have been provided to us by an industrial partner, however, we were not able to access the complete repository. The revisions are not consecutive; in the repository are up to ten other revisions between the given revisions.

An overview of the kinds of changes that have been applied to the models is given in Appendix A. For each model we created a figure that shows a histogram chart with the number of changes for each revision.

History	Precision	Recall	min. Precision	min. Recall
A	0.9989	1	0.9855	1
B	1	1	1	1
C	0.9997	1	0.9936	1
D	0.9999	1	0.9960	1
E	0.9997	0.9998	0.9994	0.9994

Table 12.3: *Results of the PR analysis summarized for each history*

History	False Pos.	False Neg.	max. Rel.	max. Sim.	avg. Rel.	avg. Sim.
A	2 (0.1%)	0	0.4250	0.8500	0.3462	0.8500
B	0	0	–	–	–	–
C	1 (0.02%)	0	0.3093	0.7625	0.3093	0.7625
D	64 (0.01%)	0	0.9000	0.9300	0.5172	0.9250
E	5	4	0.4700	0.4950	0.4200	0.4600

Table 12.4: *Overview of false results*

12.1.2 Study Results

We calculated the precision-recall values for each revision of each history. They can be found in Appendix B. In Table 12.3 we only show summarized values for each history. For each history we list the precision and recall values calculated over all identification links and we list the minimum of the precision and recall values that we calculated for each revision of the respective history. The values attest our approach to be very precise and mostly correct.

In Table 12.4 we give an insight into the false results of our approach. It shows the number of false positives and false negatives per history. Furthermore, in case of false positives we even provide the reliability and the similarity of the incorrect link with the best values and the average over all incorrectly computed links. We can see that both, the reliability and the similarity values, are very low for the incorrectly computed identification links. An exception is the Ecore model of History D; here, the best false positive has very good reliability and similarity values and even the average of similarity values of all false positives it significantly high. However, a detailed view into the links reveals that there is indeed *one* false identification link with a high reliability. The reliability values of the other false links is much lower. The high average of similarity values result from the links between elements of type *EAnnotation* or *EGenericType*. Both element types have nearly no local attributes and they are not allowed to be moved. Hence,

History	No. of links (grouped by their origin)			in % of all elements
	HASH	ITERATIVE	other	
A	5	5	0	0.5
B	0	8	0	0.7
C	0	68	0	1.7
D	0	211	0	0.03
E	0	280	0	1.4

Table 12.5: *Number of identification links with* $reliability < 0.5$ *and* $similarity < 0.85$

their similarity is always high, however, the reliability is low. Due to the best reliabilities and similarities of false identification links we recommend, that the user should inspect the links that have a reliability value below a threshold of 0.5, and a similarity value below a threshold of 0.85 respectively. Table 12.5 lists the number of identification links whose reliability and similarity values are below these thresholds. This is the number of links the user would have to review manually. However, it is not necessary that all links are inspected directly. It is sufficient if the user inspects these links when they are used in traces that are returned as the result of a concrete occurrence analysis or tracking task. The false negatives that occurred in the analysis of History E are Simulink elements of the type *Bendpoint* and *ScalarProperty*. The bend points belong to the lines that connect blocks. Although they contain only layout information they are modeled as separate elements, since the user can individually handle them in the graphical view. Under traceability aspects, however, these elements could be ignored. In the same manner we could handle the *ScalarProperty* elements. They contain the values of the attributes of blocks, since they are not directly accessible by the user they could have been ignored for tracing. For this experiment we used a mapping for the internal graph representation that was already used by SiDiff, and we did not remove unnecessary elements from the internal graph representation.

Quality of the alternative traceability information. As described before, we compared the identification links computed by our approach to the traceability information that is implied by the identifiers in the test data. For each case where the traceability information differed, we manually inspected the result and decided if either the computed link or the identifier was incorrect. The comparison of the sets of false positives and false negatives before and after the manual inspection reveals the quality of the traceability information given by identifiers. Table 12.6 lists the resulting precision and recall values for that kind of traceability. The columns *Precision* and *Recall* refer again to the values calculated for all corre-

History	Precision	Recall	min. Precision	min. Recall
A	0.9962	1	0.9067	1
B	1	0.9972	1	0.9817
C	1	0.9527	1	0.0420
D	1	0.9996	1	0.9851
E	1	0.0021	1	0.0005

Table 12.6: *Precision and Recall of the traceability information implied by identifiers*

spondences in the history, and the columns *min. Precision* and *min. Recall* refer to the worst values measured for a single revision within the history. We can see, that the traceability information implied by identifiers is mostly correct, but not in all cases. We also see that the recall value for identifier-based traceability is much lower than for our traceability approach. Hence identifiers are not sufficient to reveal all correspondences. It fails even totally for the histories C and E, which contain derived identifiers.

12.2 Study of Applicability

We performed another case study to evaluate the applicability of our approach. We tested whether the approach can be used to solve typical analysis tasks regarding evolving models. Therefore, we have build an analysis tool upon our approach [59, 147]. The tool provides a visualization of the history of a model (see Figure 12.1) and offers functionality to trace single model elements. Traces are visualized in an abstract representation that is independent of any model type. Rectangles represent different revisions of the given model, inside a rectangle each model element is represented by a small colored circle. The color gives additional information depending on the current analysis task. On the right hand side an outline view shows a list of all revisions and their elements inside. Both the graphical representation and the outline view allow developers to select model elements. Tool tips show further information about the elements. Filters can reduce the set of displayed elements. The panel on the lower part offers different analysis tasks to choose from.

The tool enables four kinds of analyses: the identification of elements across all revisions, the tracing of bugs, logical coupling analysis, and day fly analysis. The tracing of bugs has been realized as a occurrence analysis with the constraint that the similarity of the elements has to exceed a user-defined threshold. The logical coupling analysis checks how often model elements are changed together with

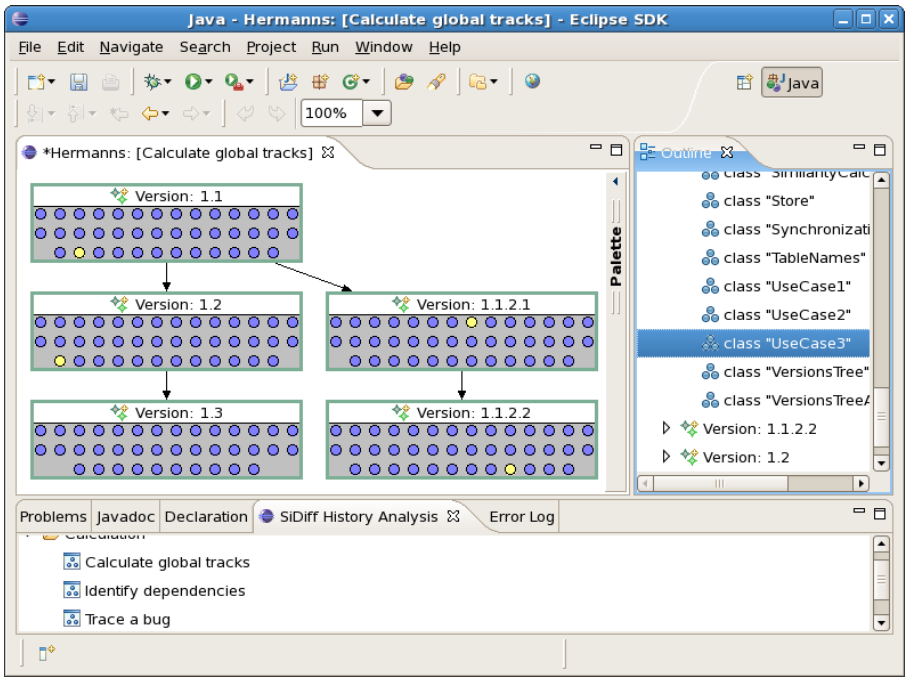
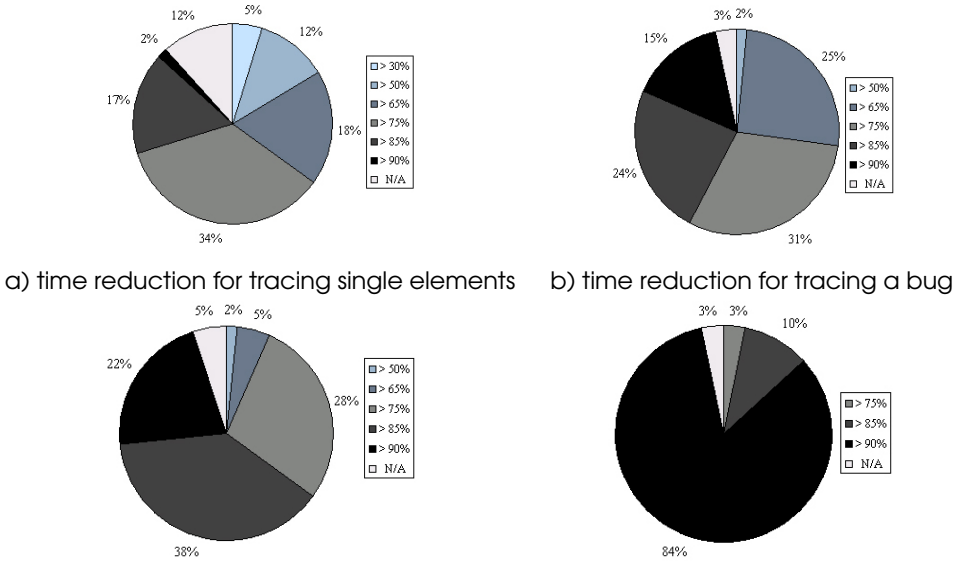


Figure 12.1: *Screenshot of the analysis tool used for the study of applicability*

other elements. It is basically the approach described in [156] applied to models. Day fly analysis reports the elements which exist only in one revision, since they are an indicator for the quality of the development process.

For each kind of the analyses mentioned above, we evaluated whether exemplary analysis tasks can be solved faster and more reliable if the prototype was used. The case study involved 30 developers with different levels of experience, mainly students and university researchers. It turned out that the evaluation results did not differ significantly among the developer types; hence, we did not differentiate between these groups. Nevertheless, detailed results can be found in [59].

The attendees got a short introduction into the analysis tool before each test started. During the test they had to analyze model histories; first manually and afterwards with the help of the tool. In order to manually analyze the model histories, test persons were provided with standardized XMI files which could be opened in a modeling tool of the test persons' choice and with JPEG files showing the graphical representation of the models. In both phases they had to fill in a questionnaire that asked for time exposure, experiences, preferences, and problems.



a) time reduction for tracing single elements

b) time reduction for tracing a bug

c) time reduction for dependency analysis

d) time reduction of day fly analysis

Figure 12.2: *Performance enhancement*

Application to an unknown history. All test persons had to analyze a history of an unknown UML class model. This is a typical task for reverse engineers. The class models represented the data model of the analyzed prototype at different development stages. The size of the single model documents ranged between 25 and 30 classes. Although that size is rather small for analysis tasks in daily practice, the different results between manual and tool-assisted analysis are significant.

For each feature of our tool implementation two specific analysis problems were given to the test persons. First point of interest has been the performance comparison of tool-based analysis vs. manual work. Figure 12.2 depicts the enhancement of performance. As shown in part (a) of the figure the time needed to trace single elements was already reduced by at least 50% for 83% of the test persons. The tracing of bugs, i.e. tracing of several elements at a time regarding the degree of changes, was reduced by at least 75% or more in almost 75% of the cases (see part (b)). In dependency analysis (part (c)) the needed time was halved for 95% of the test persons. Day flies were nearly impossible to be determined by the test persons manually as the performance enhancement states in part (d) of the figure.

Besides time reduction, the tool-based analysis produced all results correctly, whereas the test persons produced erroneous results during their manual analysis. Although the correctness was not considered interesting and has not been recorded, we estimate an error rate of 30% for the manual approach.

Summarized over the four scenarios the developers preferred significantly the

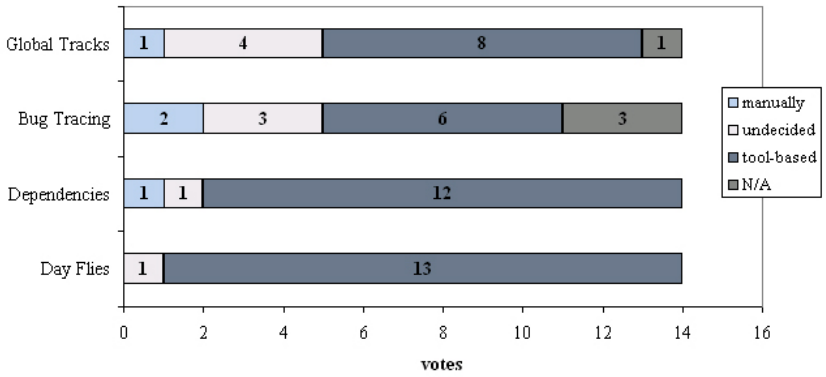


Figure 12.3: *Analysis of well-known histories*

tool solution with 108 votes. Only one vote was given to the manual approach, while there were 11 abstentions. These preferences have been explained by several reasons; e.g. performance was mentioned 59 times, 31 test persons commended simplicity. Only one participant of the study believed more in his own experience than in any tool.

Application to a well-known history. Half of the test persons also analyzed a history of UML class models that has been developed by them. The models contained around 20 classes and described the design of an auction and trading system that has been developed in groups of 4–6 persons during a one term software development course. Three student teams attended the case study.

Despite the joint development of the models and the fairly good knowledge of their history, 86% of the test persons preferred the tool-assisted analysis of the model history. Already models with 20 classes are too large to keep an overview of all elements. Figure 12.3 shows the performance advantages of the tool solution against the manual approach. While the manual approach benefits from knowledge and experiences of developers, the technical solution was superior with performance and correctness on the one side and overview, visualization, and user assistance on the other side.

Besides the normal experiments, the latter group of test persons offered the opportunity to verify the information computed by our analysis tool. The knowledge about the real history of the models allowed a thorough examination of the results computed by our tracing approach. Taken together 100% of the provided information has been judged to be correct. That result coincides with the results of the controlled experiments presented in Section 12.1.2.

12.3 Example Applications

Our approach has further been used in two other analysis tools (i.e. in addition to the prototypes presented before). The tools deal with evolution analysis. In addition, we integrated the difference metrics approach in a separate tool that enables the visualization of differences between very large models.

Scalable difference visualization. The difference metrics allow us to represent even very large differences in compact numbers. We have developed an scalable difference visualization [146]. The tool computes the difference between two revisions or different models and displays the result as polymetric views [78]. The computation of the difference is carried out by SiDiff. The metrics are computed by the metrics component of our tracing prototype. As said earlier in Section 9.3, the computation of difference metrics is independent from any traceability scenario. Hence we were able to use the metrics component as a separate component. Thereby, the difference visualization tool does not rely on the tracing approach. The visualization of the difference as a polymetric view represents elements and their relationships as a graph of rectangles connected by lines. Up to five metrics can be encoded to each element. They determine the position (x and y coordinate), the height, the width, and the color of an element's rectangle. We extended the visualization so that the border color can encode a sixth metric of an element. The visualization allows us to comprehend the changes between large model revisions. We can clearly point out the location of changes in a model, we can measure the amount of change, and we can distinguish the relevance of changes. The tools has been realized as a plugin for the Eclipse IDE. A screenshot is shown in Figure 12.4.

Fine-grained analysis of model evolution. The *FAME* tool focuses on the analysis of evolving models [148]. It provides different visualizations of the history of a model and can measure software metrics and difference metrics. It is basically a combination of the tracing tool used in our case study for applicability (see Section 12.2) and the difference visualization tool presented above. Model elements, even anonymous elements, can be traced along their history, and their evolution can be measured and visualized. The tool provides all analysis functions that have been implemented in the tracing tool. Furthermore, it enables the analysis of metrics over time. Again polymetric views are used. We draw the revisions tree of a history as polymetric view and we encode different metrics onto the revisions. Hence, the tool allows us to capture the evolution of a model.

Technically, the FAME tool has been realized upon an early prototype imple-

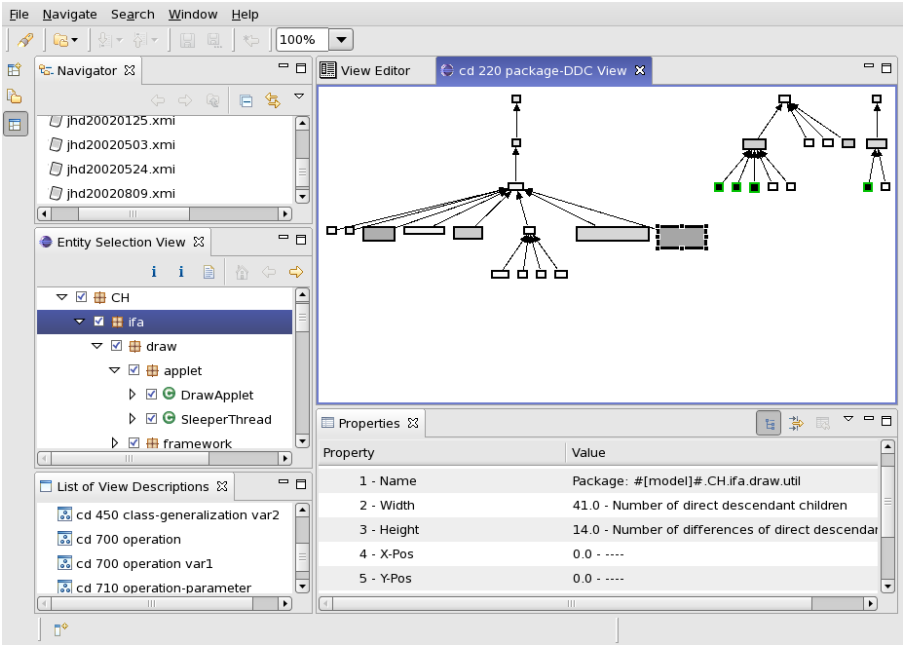


Figure 12.4: *Screenshot of the PV4E tool*

mentation of our tracing approach that was not as sophisticated as the version presented in Section 11.1. The history representation only contained identities, i.e. identification links were seen as temporary data only and quality attributes such as the reliability did not exist. Information about evolution was assigned to the versioned elements directly and thus more difficult to access. Tracing was enabled by directly querying the underlying database. In equal manner we were able to access the information about evolution. Nonetheless the tracing aspects of the FAME tool enhanced the analysis of model evolution. It attested the applicability of our tracing approach, which has later been encapsulated in a service interface to enable its usage in arbitrary tools.

The Software Evolution Toolkit. The *Software Evolution Toolkit* (SEV) is a framework for visual analysis of software evolution. It is a completely new analysis tool based on Eclipse. It can compute many different software metrics that can be visualized in different views (see Figure 12.5). The views can be combined in order to visually analyze correlations between entities based on their metrics. Particularly the analysis of metrics over time is supported. The tool uses the most recent OSGi-based service implementation of our approach (see Section 11.1) to trace the elements whose metrics are analyzed. The SEV toolkit is not limited to histories of models. It is rather generic and it has successfully been used to analyze the evo-

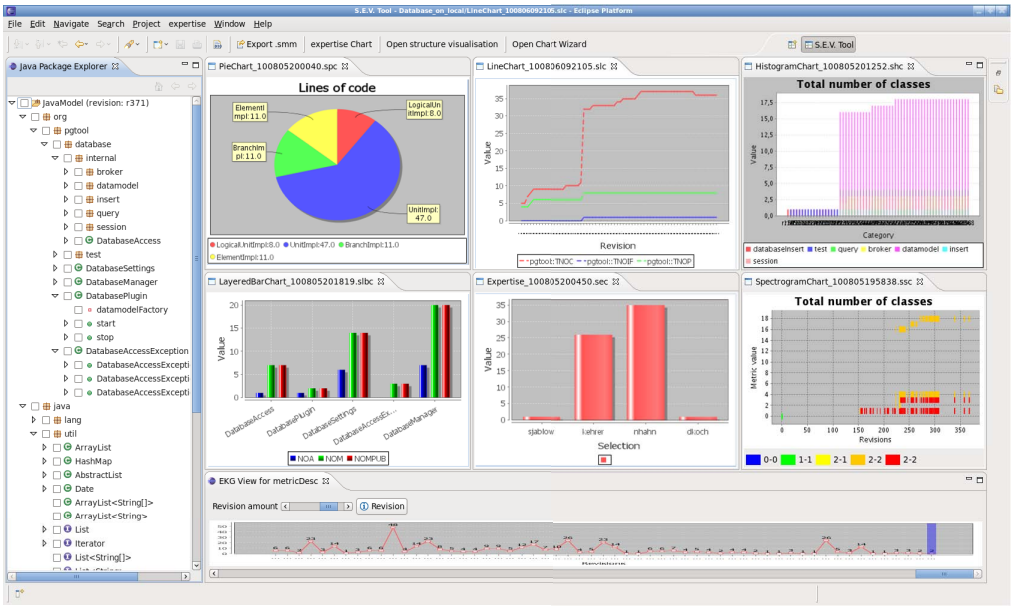


Figure 12.5: *Screenshot of the SEV tool*

lution of Java software. Thus, the tool attests the usability of our tracing service as separate component in a given environment. Furthermore, it shows that our approach is not necessarily limited to models, but it can be applied even to source code.

The examples mentioned above show that our approach was successfully used to realize analysis tools that require the traceability over time. Without the trustworthy identification of fine-grained elements the tools would not be able to compute the needed evolution data. The tool for scalable difference visualization does further show the applicability of difference metrics in other scenarios.

Part V

Epilogue

Chapter 13

Conclusions and Outlook

Model-driven engineering is a widely accepted methodology in software engineering. The portion of model-driven developed software systems is increasing rapidly. At the same time, traceability (i.e. the ability to retrace the complete engineering process) is a very important success factor for many software projects, which is sometimes even mandated by standards or norms. So far, however, the ability to trace model elements over time has often been neglected. As a consequence, traceability in model-driven engineering is often impeded as we cannot identify model elements across evolution. This problem is called the identification problem. This thesis has presented an approach to solve the identification problem. We summarize the most important characteristics of our solution in Section 13.1. In Section 13.2 we discuss the limitations of our approach, and we present ideas for improvements and future work in Section 13.3.

13.1 Discussion

Traceability in model-driven engineering is often hampered by the tracing across evolution, because we are not able to trustworthily identify model elements across the different revisions of a model. Most modeling languages, and the serialization formats of most models respectively, offer local identifiers for addressing model elements. An identifier is either an artificially generated value assigned to the element or it is a value that can be derived from the local properties of the element. These identifiers are rarely persistent, so that each revision of a model is assigned with a new set of identifiers. Existing approaches to make identifiers persistent are not sufficient in daily practice. They narrow the designers in their choice of modeling tools and they are tied to imminent risks so that parallel collaborative work is not recommendable.

In order to solve the problem of identifying model elements across different revisions, we established a representation for describing the history and the evolution of an analyzed model (i.e. the *history*). It represents all revisions of the analyzed

model and does further contain representatives for each model element of each revision. Besides describing the content of a history, it can hold information about the correspondences between the elements of subsequent model revisions and about their evolution. Such a correspondence is called an *identification link* and expresses that the connected model elements are representations of the same element at different times. Identification links that connect representations of the same element form in turn the *identity* of a model element. The identity can be seen as a globally unique identifier that allows us to identify an arbitrary model element within the whole history. Furthermore, the identification links contain information about the changes that have been applied to a model element from one revision to another. In summary, they describe the evolution of the model over time.

The identification links are computed on the basis of a model comparison algorithm. Model comparison deals with the computation of differences between two model revisions. The key problem is the computation of a matching that maps the elements of one revision to the elements of the other revision. Each pair of elements represents the same element at different times, and the changes applied to this element can be deduced. We use such a matching algorithm to compute a matching between two subsequent revisions of the analyzed model. If the computation is applied to all pairs of subsequent revisions, we get identification links along the complete history, i.e. the identities.

We use the similarity-based comparison algorithm *SiDiff* that can be configured to arbitrary types of models. Our approach is thus not limited to particular modeling languages. We are also able to use other similarity-based approaches for the computation of traceability information. We extended the comparison approach so that we are able to assess the reliability of the found correspondences which in turn enables the assessment of the reliability of the derived traceability information. Our approach can be applied to existing model histories that are for instance managed in a configuration management system, but the computed information can always be extended if new revisions of a model are created.

The traceability information computed by our approach can be used to locate the occurrences of a given model element in other revisions. Besides occurrence analysis, we enable the tracking of model elements over time. We can follow a model element along its evolution backward to the creation of the model element as well as forward up to the deletion of a model element. we can even follow elements that have temporary been deleted. The occurrence analysis and the tracking can further be enriched with the definition of constraints that the traced elements have to fulfill. Besides tracing of single elements, we can even trace fragments of a model. Therefore, we trace each element of that fragment individually and

check whether the relationships between the elements of the source revision do also exist in the target revision.

The change information that is assigned to identification links allows the user to capture the evolution that has been applied to the traced elements. Since the description of evolution as a long list of single changes is not concise and the user would drown in a plethora of information, we developed *difference metrics* that allows us to express the amount of changes in numbers. We can count different types of changes for the different types of model elements. The computation is independent from concrete model types. Besides that, we enable the computation of more specific metrics by parameterization. We can take model type specific aspects into account and can, for instance, classify the changes according to their relevance. The result is a set of metric values that is easier to overview than the set of concrete changes, and coherency or outliers can be detected much faster.

In order to validate our approach we have prototypically implemented it as a service for the OSGi platform. All traceability information, the results of the model comparison, and the computed metrics are stored in a relational database and they can be accessed through an object-oriented Java interface. Both technologies enable the integration of our approach into other tools. We have also implemented an interactive tracing tool as another prototype. The tool uses the service implementation of our approach and provides a graphical user interface to trace model elements and fragments and to query the history and the evolution data. We further integrated the approach into two evolution analysis tools. This integration attested us the applicability of our approach.

In addition, we evaluated the correctness of the computed identification links by means of controlled experiments. We analyzed the histories of different models and compared the results with alternative traceability information. The experiments revealed very good results that certify the quality of our approach. The precision of our approach was 98.5% in the worst case. The recall was 99.9% or better. The results of our tracing approach are thus better than traceability information expressed by persistent identifiers. The high quality of our approach enables the application in industrial practice.

13.2 Limitations

Despite the correctness and the applicability of our approach, two cases exist that cannot be handled sufficiently. Some changes applied to the analyzed model cannot be traced, because the model comparison that is the basis of our approach does not detect them reasonably.

Copies of elements. Some modeling tools provide the ability to copy a model element or a set of model elements. In a UML model, for example, we can copy a class with all its attributes and operations; we just have to provide a new name or a new namespace for the copy. As a consequence duplicates can exist. If we trace the first version of the class to the revision that contains the class and its copy, we could argue that there should be two identification links. They express that both classes have their origin in the class that was copied. So far, we just support the tracing along identities, i.e. we create the link between the class in the first revision and the same class in the second revision. In the strict sense, the copy of the class is a *new* class. An argument for this behavior is the fact that the copy is minimally changed, since it has been renamed. For the attributes and operations of that class we cannot argue likewise. The attributes and operations are exact copies of the originals. Again, we would just create one identification link for each attribute and operation, i.e. the one expressing the identity. The copies are not traced to previous revisions, because they are new. However, in practice it would be useful if we could also trace the copies to the original elements.

For this case, we recommend creating an additional identification link between the element that has been copied and the copy. The identification link points from the revision without the copy to the revision with the copy; it is not within a single revision. The origin attribute of this link should be set to *copy detection* to distinguish the link from actual identification links. Subsequently, we call such a link a *copy link*.

The current model comparison approaches (including SiDiff) do not support the detection of copy operations yet. Hence, we have to infer this information from the comparison result. One possible approach for locating copied elements could be the analysis of the model elements that were reported by the model comparison to be structurally new. Such an element could be a copy of another element if the following rules are fulfilled:

- The type of the element has to be configured so that it can be copied or the container element is already assigned with a copy link. Thereby, we can prevent the creation of copy links between primitive elements. For example, the parameters of operations are not seen as copies of each other, although they are named and typed equally. They are only understood as copied if the operation has been copied.

- There is a model element in the ancestor revision that has an equal hash value or a similarity exceeding a predefined threshold. In the latter case all contained elements must in turn have the same hash values or adequate

similarities compared to the elements contained by the potential copy.[1] If such an element exists, it could be the original element that has been copied. The thresholds can be configured differently for each type of model element.

- The original element has to exist in the revision without the copy and in the revision with the copy. Both representations have to correspond.

The rules mentioned above define the necessary condition for an element to be the copy of another element. However, it is not a sufficient condition, since many other arbitrary constraints might exist. For example, it is not possible in UML models to copy an association unless it is copied together with the classes that are connected by the association. Hence, the detection of copy operations is still an unsolved problem.

Refactorings. A problem that is closely related to the copy problem mentioned above is the detection of refactorings or other complex operations. A complex operation is a single edit operation provided in the modeling tool that leads to many changes in the model. An example is the *create subsystem* operation in MATLAB/Simulink™. It allows the user to select a set of blocks and automatically creates a new subsystem by moving all selected blocks into that subsystem. A refactoring is similar to a complex operation, however, it often leads to even more changes in the model than a complex operation. Furthermore, the changes of a complex operation are often regional, whereas refactorings may affect the whole model.

Some complex operations and refactorings do not affect the result of our tracing approach. In case of the *create subsystem* operation for example, we can trace all the moved blocks. The only problem is the comprehensibility by the user. In many cases the user is not aware of the single changes applied by a complex operation. The assessment of the evolution might be misleading. The user performed one edit operation on the model, but the difference metrics would report many changes.

However, there are refactorings and also some complex operations that hamper our tracing approach. An example the refactoring "extract superclass" [45]. It creates a new superclass for classes that share many attributes or operations, and the common attributes and operations are moved to it. Here we have the problem, that multiple elements of one revision have to correspond to a single element in the other revision. That requires n-ary correspondences that are not supported by the current model comparison approaches. Furthermore, it is a problem that the classes from which the superclass has been extracted cannot be found as corresponding if the bigger part of a class was moved to the superclass.

[1] In case of an equal hash value all contained elements are equal per definition.

The old and the new version of that class would be too different to exceed the similarity threshold.

Although the n-ary correspondences are not yet reported by the model comparison, they are supported by our approach. The creation of identification links is independent from the number of correspondences found for an element, since we create one identification link for each correspondence. Our history data model allows us to store multiple identification links for a versioned element. It has not to be changed if future versions of the model comparison will report n-ary correspondences.

13.3 Outlook

In the previous section we have discussed two problems of the model comparison which lead to a limited applicability of our approach to histories that contain particular types of changes. Nonetheless, our tracing approach grants access to solutions of a broad range of research problems.

Supported research. Our approach enables a thorough traceability in model-driven engineering processes. Besides traceability along transformations (see Section 2.3.2), we can now even trace along manual changes. This allows us to solve other research problems such as incremental transformations. So far, a manual change to the input model of a transformation chain required the re-execution of the complete chain. The tracing along manual edits will allow us to perform incremental transformations, so that only parts of the transformation chain have to be re-executed. There are still unsolved problems such as the sufficient patching of partial modifications onto the later results of a transformation chain [75], however, the traceability and the capturing of the evolution was obviously a significant problem that we have smoothed out.

Due to the trustworthy identification of model elements over time, we are for the first time able to thoroughly analyze the evolution of models in repositories. The FAME tool (see Section 12.3) is the first step into that direction, however, compared to the research applied to evolution of code bases of software systems [133] there are still many opportunities. With the ability to research the evolution of models comes the opportunity that modeling tools and repositories can better be adapted to the needs of users. Especially the configuration management tools for models (i.e. diff and merge) could be improved if we can capture “the typical” model evolution. Furthermore, the approaches to model repositories presented so far required certain modeling tools or they constrained the set of supported models (see Section 2.1.2.2). Thus the applicability of model-driven approaches

was often seen suspiciously by practitioners. The identification over time can be used to close this gap.

Transfer to other domains. In this thesis we have extensively referred to model-driven engineering, i.e. the models are used to develop the software system. However, models are also used for other purposes. In reverse-engineering, for example, they are inferred from an existing software system to better comprehend it. Similar is the field of software evolution analysis. Especially if the software is not given as source-code, but has to be decompiled from machine code, expressive identifiers are often missing. The application of our tracing approach could help to identify the fine-grained entities of one revision in another. The SEV Toolkit, which uses our solution, is a good example for applying our traceability approach to source code. However, the application to reverse-engineered models should be investigated in more detail. It would especially be interesting to quantitatively analyze to what extent the temporal distance between the reverse-engineered snapshots influences the quality of the identification.

Furthermore, models are used in completely different domains such as bioinformatics. The models can for instance represent the metabolism of cells. The atoms and molecules, which are modeled here, do not have an identity. Our approach could be applied to enable the identification of such elements over time. The applicability of our solution to another domain obviously depends on the applicability of SiDiff to that domain or the existence of an alternative comparison approach for models of that domain. A first attempt to apply SiDiff to the comparison of molecular graphs can be found in [55]. However, it is not clear whether our traceability approach can be applied to such domains without changes. The additional requirements that are implied by the different domains would have to be explored.

Improvements. The limitations discussed before are obviously a starting point for future improvements of our approach.

The ideas of detecting copy links should definitively be evaluated in more detail. We have already tried to evaluate the above-mentioned rules to detect copied elements, however, formulating the domain-specific constraints is not trivial. There are many aspects to be considered if we want to prohibit the case that each element is considered to be the copy of another element.

The support for complex operations and refactorings is also not a trivial problem. Each modeling language comes with its own definitions of such operations. This is a separate research problem that must be solved. If there will be a solution to that problem, our approach can still be applied without significant changes. The procedure of creating identification links has not to be changed, however, the

computation of the difference metrics requires some adjustments, since new types of changes will be reported by the model comparison.

Besides the improvements in model comparison, which allow us to compute tracing information even if the models have been refactored, we can also improve the quality of the computed traceability information by so-called *shortcut links*. Therefore, we also compare every x-th revision directly, so that we get additional links connecting the elements that are otherwise connected by x identification links spanning over the intermediate revisions.[2] These links build shortcuts. The identification of elements between distant revisions *can* thereby become more reliable, because they would be connected with shorter paths of identification links. However, it is not clear whether the reliability is increased by shortcut links, because neither the reliability nor the similarity fulfill the triangle inequality. The similarity values that result from the direct comparison of two distant revisions can be higher than the similarity values resulting from the comparison of the intermediate revisions if changes of an early revision have been made undone in a later revision. The reliability values are calculated from many factors, e.g. the similarity or the correspondences of the neighborhood, the triangle inequality is thereby unfulfilled due to the used similarity values, but also the neighborhood can change arbitrarily. It is thus not clear if a shortcut link leads to better results in the identification. In future work, we should aim for a further investigation of that problem.

In addition, the computation of reliability values can probably be improved by integrating it into the model comparison algorithms. So far, we attached the computation from outside to the matching algorithm. The computation might become more efficient and more reliable if we directly integrate it into the algorithm. In SiDiff, for instance, the similarity would no longer be the only criterion for matching elements. If it is based on the correspondences of other elements, the reliability of these correspondences could be considered for the match decision. If we further consider, that correspondences could depend on each other, we could research if the explicit management of these dependencies leads to better results, especially if the user manually revokes decisions of the algorithm.

It would further be interesting to apply statistical methods to analyze reliabilities, so that we do not need to inspect the values manually by visualization. In combination with a study of typical evolution profiles (as enabled by the repository mining we mentioned before) we could better assess the quality of traceability information.

The prototype implementation can also be improved. As described before, the data model that we use to describe the history and to store all tracing informa-

[2] x is a configurable parameter.

tion has been realized as an object-relational wrapper, since all data is physically stored in a relational database. The high number of database accesses is a bottle neck in the performance of creating the identification links and traces. However, keeping all data in memory is also not sufficient due to heap limitations. Although the performance is not a critical factor for our approach, because the traceability information is only computed once and can be performed overnight, it is feasible to investigate the bottle necks in more detail and to eliminate them.

Bibliography

[1] European Conference on Model-Driven Architecture - Traceability Workshop Series (ECMDA-TW), 2005-2009.

[2] Special Issue on Traceability in Model-Driven Engineering. *Software and Systems Modeling*, 9(4), 2010.

[3] Netta Aizenbud-Reshef, Brian T. Nolan, Julia Rubin, and Yael Shaham-Gafni. Model Traceability. *IBM Systems Journal*, 45(3):515–525, 2006.

[4] Marcus Alanen and Ivan Porres. Difference and Union of Models. In *LNCS vol. 2863, UML 2003 - The Unified Modeling Language*, pages 2–17. Springer, October 2003.

[5] Bastien Amar, Herve Leblanc, and Bernard Coulettee. A Traceability Engine Dedicated to Model Transformation for Software Engineering. In *Proc. of the ECMDA Traceability Workshop (ECMDA-TW)*, 2008.

[6] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, October 2002.

[7] Giuliano Antoniol, Yann-Gaël Guéhéneuc, Ettore Merlo, and Paolo Tonella. Mining the Lexicon Used by Programmers during Sofware Evolution. In *Proc. of the 23rd International Conference on Software Maintenance (ICSM'07)*, pages 14–23, October 2007.

[8] Paul Arkley and Steve Riddle. Overcoming the Traceability Benefit Problem. In *Proc. of the 13th International Conference on Requirements Engineering (RE'05)*, pages 385–389, 2005.

[9] Wayne A. Babich. *Software Configuration Management: Coordination for Team Productivity*. Addison-Wesley, 1986.

[10] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[11] Mikael Barbero, Marcos Didonet Del Fabro, and Jean Bézivin. Traceability and Provenance Issues in Global Model Management. In *Proc. of the ECMDA Traceability Workshop (ECMDA-TW)*, pages 47–55, 2007.

[12] Ivo Barone, Andrea De Lucia, Fausto Fasano, Esterino Rullo, Giuseppe Scanniello, and Genoveffa Tortora. COMEOVER: Concurrent Model Versioning. In *Proc. of the International Conference on Software Maintenance (ICSM'08)*, 2008.

[13] Lars Bendix and Pär Emanuelsson. Diff and Merge Support for Model Based Development. In *Proc. of the 2008 International Workshop on Comparison and Versioning of Software Models (CVSM'08)*, pages 31–34, May 2008.

[14] Keith H. Bennett and Vaclav T. Rajlich. Software Maintenance and Evolution: a Roadmap. In *The Future of Software Engineering*, pages 75–87. ACM Press, 2000.

[15] Bernd Bruegge, Allan H. Dutoit, and Timo Wolf. Sysiphus: Enabling Informal Collaboration in Global Software Development. In *Proc. of the First International Conference on Global Software Engineering (ICGSE'06)*, October 2006.

[16] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a Taxonomy of Software Change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–322, September 2005.

[17] Jason Jen-Yen Chen and Shih-Chien Chou. Consistency Management in a Process Environment. *Journal of Systems and Software*, 47(2-3):105–110, July 1999.

[18] Jane Cleland-Huang, Carl K. Chang, and Mark Christensen. Event-Based Traceability for Managing Evolutionary Change. *IEEE Transactions on Software Engineering*, 29(9):796–810, 2003.

[19] Pietro Colombo, Vieri Del Bianco, and Luigi Lavazza. Fine-Grained Integrated Management of Software Configurations and Traceability Relations. In *Proc. of the 3rd International Conference on Software and Data Technologies (ICSOFT'08)*, pages 159–164. INSTICC Press, 2008.

[20] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[21] Alistair Cookburn. *Agile Software Development*. Addison-Wesley, 2001.

[22] Marco Costa and Alberto R. Da Silva. RT-MDD Framework – A Practical Approach. In *Proc. of the ECMDA Traceability Workshop (ECMDA-TW)*, pages 17–26, 2007.

[23] Andrea De Lucia, Fausto Fasano, Rita Francese, and Genoveffa Tortora. ADAMS: an Artefact-Based Process Support System. In *Proc. of 16th International Conference on Software Engineering and Knowledge Engineering (SEKE'04)*, pages 31–36, 2004.

[24] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. ADAMS Re-Trace: a Traceability Recovery Tool. In *Proc. of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 32–41, 2005.

[25] Andrea De Lucia, Fausto Fasano, Giuseppe Scanniello, and Genoveffa Tortora. Concurrent Fine-grained Versioning of UML Models. In *Proc. of 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*. IEEE Computer Society Press, March 2009.

[26] Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz. Finding Refactorings Via Change Metrics. In *Proc. of the 15th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'00)*, October 2000.

[27] Nicholas Drivalos, Richard F. Paige, Kiran J. Fernandes, and Dimitrios S. Kolovos. Towards Rigorously Defined Model-to-Model Traceability. In *Proc. of the ECMDA Traceability Workshop (ECMDA-TW)*, 2008.

[28] Jürgen Ebert and Angelika Franzke. A Declarative Approach to Graph Based Modeling. In *LNCS vol. 903, Proc. of the 20th Intl. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 38–50. Springer, 1994.

[29] Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph Technology in Reverse Engineering: The TGraph Approach. In *LNCS vol. 126, Workshop Software Reengineering*, pages 67–81. GI, 2008.

[30] Eclipse Foundation. EMF Compare. `http://wiki.eclipse.org/index.php/EMF_Compare`, 2009.

[31] Eclipse Foundation. The Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/`, 2009.

[32] Eclipse Foundation. Eclipse Graphical Editing Framework. `http://www.eclipse.org/gef/`, 2010.

[33] Eclipse Foundation. EMF Connected Data Objects (CDO). http://wiki.eclipse.org/index.php/CDO, 2010.

[34] Eclipse Foundation. Rich Client Platform. http://www.eclipse.org/rcp/, 2010.

[35] Eclipse Foundation. The Eclipse Project. http://www.eclipse.org/eclipse/, 2010.

[36] Eclipse Foundation. The MDT/UML2 Project. http://www.eclipse.org/uml2/, 2010.

[37] Eclipse Foundation. The Standard Widget Toolkit. http://www.eclipse.org/swt/, 2010.

[38] Alexander Egyed. A Scenario-Driven Approach to Trace Dependency Analysis. *IEEE Transactions on Software Engineering*, 29(2):116–132, February 2003.

[39] Epsilon Community. The Epsilon Model Management Platform. http://www.eclipse.org/gmt/epsilon/, 2008.

[40] ETAS. ASCET Software Products. http://www.etas.com/en/products/ascet_software_products.php, March 2009.

[41] Jean-Rémy Falleri, Marianne Huchard, and Clémentine Nebut. Towards a Traceability Framework for Model Transformations in Kermeta. In *Proc. of the ECMDA Traceability Workshop (ECMDA-TW)*, 2006.

[42] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Cregut, and Marc Pantel. The TOPCASED Project: a Toolkit in Open Source for Critical Aeronautic Systems Design. In *Proc. of the 3rd Embedded Real Time Software Conference (ERTS'06)*, January 2006.

[43] Jean-Marie Favre. Languages Evolve Too! Changing the Software Time Scale. In *Proc. of the 8th International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 33–44. IEEE Computer Society, September 2005.

[44] Karl Fogel and Moshe Bar. *Open Source Development with CVS*. Paraglyph Press, June 2003.

[45] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[46] Free Software Foundation. GNU Diffutils. `http://www.gnu.org/software/diffutils/`, 2009.

[47] Fujaba. Project Homepage. `http://www.fujaba.de`, December 2008.

[48] Phillip B. Gibbons, Richard M. Karp, Gary L. Miller, and Danny Soroker. Subtree Isomorphism is in Random NC. In *LNCS vol. 319, VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, (AWOC'88)*, pages 43–52, New York, June/July 1988. Springer-Verlag.

[49] Martin Girschick. UMLDiff: Erkennung und Analyse von Unterschieden in Klassendiagrammen und Sequenzdiagrammen. Diploma Thesis (German), Technical University of Darmstadt, 2002.

[50] Flori Glitia, Anne Etien, and Cedric Domoulin. Traceability for an MDE Approach of Embedded System Conception. In *Proc. of the ECMDA Traceability Workshop (ECMDA-TW)*, pages 27–37, 2008.

[51] Michael W. Godfrey and Qiang Tu. Tracking Structural Evolution Using Origin Analysis. In *Proc. of the International Workshop on Principles of Software Evolution (IWPSE'02)*, May 2002.

[52] Michael W. Godfrey and Lijie Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Transactions on Software Engineering*, 31(2), February 2005.

[53] Guy-Dominic Gorek. Untersuchungen zum Abgleich vager Modelle in der Systemanalyse. Diploma Thesis (German), University of Siegen, 2010.

[54] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An Analysis of the Requirements Traceability Problem. In *Proc. of the First International Conference on Requirements Engineering*, pages 94–101, 1994.

[55] Oliver Grassow. Vergleich molekularer Graphen mit Hilfe des SiDiff-Algorithmus. Diploma Thesis (German), University of Siegen, 2009.

[56] Jeff Gray, Juha-Pekka Tolvanen, Steven Kelly, Aniruddha Gokhale, Sandeep Neema, and Jonathan Sprinkle. Domain-Specific Modeling. In Paul A. Fishwick, editor, *Handbook of Dynamic System Modeling*. Chapman & Hall, May 2007.

[57] Maciej Hapke, Andrzej Jaszkiewicz, Krzysztof Kowalczykiewicz, Dawid Weiss, and Piotr Zielniewicz. OPHELIA - Open Platform for Distributed Software Development. In *Open Source for an Information and Knowlegde Society: Proc. of the Open Source International Conference*, 2004.

[58] Jane Huffman Hayes, Alex Dekhtyar, and James Osborne. Improving Requirements Tracing via Information Retrieval. In *Proc. 11th International Requirements Engineering Conference (RE'03)*, pages 151–161. IEEE Computer Society, 2003.

[59] Hermann Hutter. Nachverfolgbarkeit von Modellelementen in Versionshistorien. Diploma Thesis (German), University of Siegen, 2007.

[60] IBM Corporation. Rational Software Architect. `http://www-01.ibm.com/software/rational/products/swarchitect/`, 2010.

[61] IEEE Computer Society. International Conference of Program Comprehension. `http://www.program-comprehension.org/`.

[62] IEEE Computer Society. International Conference of Software Maintenance. `http://conferences.computer.org/icsm/`.

[63] IEEE Computer Society. Software Engineering Terminology – Standard 610.12, 1990.

[64] IEEE Computer Society and The Reengineering Forum. Working Conference on Reverse Engineering. `http://reengineer.org/`.

[65] International Organization for Standardization. Draft International Standard ISO/DIS 26262-1 (Road Vehicles –Functional Safety–), December 2009.

[66] Frederic Jouault. Loosely Coupled Traceability for ATL. In *Proc. of the ECMDA Traceability Workshop (ECMDA-TW)*, pages 29–37, 2005.

[67] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proc. of the Model Transformations in Practice Workshop at MoDELS*, October 2005.

[68] Timo Kehrer and Edmund Ihler. Process-Integrated Refinement Patterns in UML. In *Proc. of the 21st International Conference on Software & Systems Engineering and their Applications (ICSSEA'08)*, 2008.

[69] Udo Kelter and Maik Schmidt. Comparing State Machines. In *Proc. of the 2008 International Workshop on Comparison and Versioning of Software Models (CVSM'08)*, May 2008.

[70] Udo Kelter, Jürgen Wehren, and Jörg Niere. A Generic Difference Algorithm for UML Models. In *Proc. of Software Engineering*, pages 105–116, March 2005.

[71] Stuart Kent. Model Driven Engineering. In *Proc. of the 3rd International Conference on Integrated Formal Methods*, pages 286–298, 2002.

[72] Maximilian Kögel. Towards Software Configuration Management for Unified Models. In *Proc. of the 2008 International Workshop on Comparison and Versioning of Software Models (CVSM'08)*, 2008.

[73] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. On-Demand Merging of Traceability Links with Models. In *Proc. of the 2nd ECMDA Traceability Workshop (ECMDA-TW)*, 2006.

[74] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model Comparison: a Foundation for Model Composition and Model Transformation Testing. In *Proc. of the First International Workshop on Global Integrated Model Management co-located with ICSE (GAMMA'06)*, May 2006.

[75] Patrick Könemann. Model-Independent Differences. In *Proc. 2nd Workshop on Comparison and Versioning of Software Models (CVSM'09)*, pages 37–42. IEEE Press, May 2009.

[76] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Model-Based DSL Frameworks. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 602–616, October 2006.

[77] Ivan Kurtev, Matthijs Dee, Arda Goknil, and Klaas van den Berg. Traceability-Based Change Management in Operational Mappings. In *Proc. of the ECMDA Traceability Workshop (ECMDA-TW) 2007*, pages 57–67, 2007.

[78] Michele Lanza and Stephane Ducasse. Polymetric Views - a Lightweight Visual Approach to Reverse Engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.

[79] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, 34(11):44–51, 2001.

[80] M.M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proc. of the IEEE - Special Issue on Software Engineering*, 68(9):1060–1076, September 1980.

[81] Yuehua Lin, Jeff Gray, and Frederic Jouault. DSMDiff: A Differentiation Tool for Domain-Specific Models. *European Journal of Information Systems*, 16(4):349–361, August 2007.

[82] Jon Loelinger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly, May 2009.

[83] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics*. Prentice Hall, 1994.

[84] Marco Lormans and Arie van Deursen. Reconstructing Requirements Coverage Views from Design and Test Using Traceability Recovery via LSI. In *Proc. of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'05)*, pages 37–42. ACM, 2005.

[85] Jochen Ludewig. Models in Software Engineering – An Introduction. *Software and Systems Modeling*, 2(1):5–14, March 2003.

[86] Patrick Mäder, Orlena C. Z. Gotel, and Ilka Philippow. Enabling Automated Traceability Maintenance by Recognizing Development Activities Applied to Models. In *Proc. 23rd International Conference on Automated Software Engineering (ASE'08)*, September 2008.

[87] Patrick Mäder, Orlena C. Z. Gotel, and Ilka Philippow. Rule-Based Maintenance of Post-Requirements Traceability Relations. In *Proc. 16th International Requirements Engineering Conference (RE'08)*, pages 23–32, September 2008.

[88] Jonathan I. Maletic, Michael L. Collard, and Bonita Simoes. An XML Based Approach to Support the Evolution of Model-to-Model Traceability Links. In *Proc. of the 3rd ACM International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'05)*, November 2005.

[89] Adrian Marcus and Jonathan I. Maletic. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In *Proc. of the 27th International Conference on Software Engineering (ICSE'05)*, pages 125–135, 2005.

[90] Akhil Mehra, John C. Grundy, and John G. Hosking. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 204–213. ACM, November 2005.

[91] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In *Proc. of the 18th Intl. Conf. on Data Engineering (ICDE)*, 2002.

[92] Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.

[93] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[94] MetaCase. MetaEdit+. `http://www.metacase.com/`, March 2009.

[95] Ethan V. Munson and Tien N. Nguyen. Concordance, Conformance, Versions, and Traceability. In *Proc. of the 3rd ACM International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'05)*, November 2005.

[96] Leonardo G. P. Murta, Correa Chessman, Joao G. Prudencio, and Cláudia M. L. Werner. Towards Odyssey-VCS 2: Improvements Over A UML-Based Version Control System. In *Proc. of the 2008 International Workshop on Comparison and Versioning of Software Models (CVSM'08)*, May 2008.

[97] Leonardo G. P. Murta, Andre van der Hoek, and Cláudia M. L. Werner. Continuous and Automated Evolution of Architecture-to-Implementation Traceability Links. *Automated Software Engineering*, 15(1):75–107, 2008.

[98] Eugene W. Myers. An O(ND) Difference Algorithm and its Variations. *Algorithmica*, 1(2):251–266, 1986.

[99] Johan Natt och Dag, Vincenzo Gervasi, Sjaak Brinkkemper, and Bjorn Regnell. A Linguistic-Engineering Approach to Large-Scale Requirements Management. *IEEE Software*, 22(1):32–39, 2005.

[100] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and Merging of Statecharts Specifications. In *Proc. of the 29th International Conference on Software Engineering (ICSE'07)*, pages 54–64. IEEE Computer Society, 2007.

[101] Tien N. Nguyen and Ethan V. Munson. A Model for Conformance Analysis of Software Documents. In *Proc. of the 6th International Workshop on Principles of Software Evolution (IWPSE'03)*, page 24. IEEE Computer Society, 2003.

[102] Tien N. Nguyen, Ethan V. Munson, and John T. Boyland. An Infrastructure for Development of Multi-Level, Object-Oriented Configuration Management Services. In *Proc. of the 27th International Conference on Software Engineering (ICSE'05)*, 2005.

[103] Tien N. Nguyen and Ethan V. Munson. The Software Concordance: A New Software Document Management Environment. In *Proc. of the 21st Annual International Conference on Documentation (SIGDOC'03)*, pages 198–205. ACM, October 2003.

[104] Object Management Group. The official MDA guide, v.1.0.1 (formal 03-06-01). http://www.omg.org/cgi-bin/doc?omg/03-06-01, June 2003.

[105] Object Management Group. Meta Object Facility Core Specification (MOF) 2.0 (formal 06-01-01). http://www.omg.org/docs/formal/02-04-03.pdf, January 2006.

[106] Object Management Group. The MOF2.0/XMI mapping, v.2.1.1 (formal 2007-12-01). http://www.omg.org/spec/XMI/2.1.1/, 2007.

[107] Object Management Group. The UML Infrastructure Specification, v.2.1.2 (formal 2007-11-04). http://www.omg.org/spec/UML/2.1.2/, 2007.

[108] Object Management Group. The UML Superstructure Specification, v.2.1.2 (formal 2007-11-02). http://www.omg.org/spec/UML/2.1.2/, 2007.

[109] Object Management Group. Common Object Request Broker Architecture (CORBA/IIOP) Specification, version 3.1. http://www.omg.org/spec/CORBA/3.1/, January 2008.

[110] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification Version 1.0. http://www.omg.org/spec/QVT/1.0/, April 2008.

[111] Object Management Group. http://www.omg.org/spec/OCL/2.2/, 2010.

[112] Dirk Ohst and Udo Kelter. A Fine-Grained Version and Configuration Model in Analysis and Design. In *Proc. of the International Conference on Software Maintenance (ICSM'02)*, pages 521–527, October 2002.

[113] Dirk Ohst, Michael Welle, and Udo Kelter. Difference Tools for Analysis and Design Documents. In *Proc. of the 19th International Conference on Software Maintenance (ICSM 2003)*, pages 13–22. IEEE Computer Society, September 2003.

[114] Dirk Ohst, Michael Welle, and Udo Kelter. Differences Between Versions of UML Diagrams. In *Proc. of the Joint Meeting of ESEC/FSE'03*, September 2003.

[115] Jon Oldevik and Tor Neple. Traceability in Model to Text Transformations. In *Proc. of the ECMDA Traceability Workshop (ECMDA-TW)*, 2006.

[116] Hamilton L. R. Oliveira, Leonardo G. P. Murta, and Cláudia M. L. Werner. Odyssey-VCS: a Flexible Version Control System for UML Model Elements. In *Proc. of the 12th International Workshop on Software Configuration Management (SCM'05)*, pages 1–16, September 2005.

[117] Oracle Corporation. MySQL Database Management System. `http://www.mysql.com/`, 2010.

[118] OSGi Alliance. OSGi Service Platform Release 4 Version 4.1. `http://www.osgi.org/Specifications/HomePage`, May 2007.

[119] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly, September 2008.

[120] Klaus Pohl. *Process-Centered Requirements Engineering*. Advances Software Development. John Wiley & Sons Ltd., 1996.

[121] Klaus Pohl. PRO-ART: A Process Centered Requirements Engineering Environment. In M. Jarke, C. Rolland, and A. Sutcliffe, editors, *The NATURE of Requiremtents Engineering*. Shaker Verlag, 1999.

[122] C. V. Ramamoorthy, Yutaka Usuda, Atul Prakash, and W. T. Tsai. The Evolution Support Environment System. *IEEE Transactions on Software Engineering*, 16(11):1225–1234, 1990.

[123] Raghu Reddy, Robert France, Sudipto Ghosh, Franck Fleurey, and Benoit Baudry. Model Composition – a Signature-Based Approach. In *Proc. of Aspect Oriented Modeling (AOM) Workshop associated to MoDELS'05*, October 2005.

[124] Jungkyu Rho and Chisu Wu. An Efficient Version Model of Software Diagrams. In *Proc. of the Fifth Asia Pacific Software Engineering Conference (APSEC'98)*, pages 236–243, December 1998.

[125] Maik Schmidt and Tilman Gloetzner. Constructing Difference Tools for Models Using the SiDiff Framework. In *Proc. of the 30th International Conference on Software Engineering (ICSE'08)*, page 947f., May 2008.

[126] Maik Schmidt, Sven Wenzel, Timo Kehrer, and Udo Kelter. History-Based Merging of Models. In *Proc. of the 2009 International Workshop on Comparison and Versioning of Software Models (CVSM'09)*, May 2009.

[127] Christian Schneider. *CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten*. PhD thesis, (in German). University of Kassel, Germany, December 2007.

[128] Christian Schneider, Albert Zündorf, and Jörg Niere. CoObRA - a Small Step for Development Tools to Collaborative Environments. In *Proc. of the Workshop on Directions in Software Engineering Environments (WoDiSEE'04)*, 2004.

[129] Hannes Schwarz, Jürgen Ebert, and Andreas Winter. Graph-Based Traceability: A Comprehensive Approach. *Software and Systems Modeling*, 9(4):473–492, 2010.

[130] Petri Selonen and Markus Kettunen. Metamodel-Based Inference of Inter-Model Correspondence. In *Proc. of 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 71–80, March 2007.

[131] Mika Siikarla and Tarja Systä. Decision Reuse in an Interactive Model Transformation. In *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR) 2008*. IEEE Computer Society, April 2008.

[132] Mike Smith, Dawid Weiss, Pauline Wilcox, and Rick Dewar. The Ophelia Traceability Layer. In *Cooperative Methods and Tools for Distributed Software Processes, 2nd Workshop on Cooperative Supports for Distributed Software Engineering Processes*, pages 150–161, 2003.

[133] IEEE Computer Society. Mining Software Repositories – Working-Conference Series. `http://msrconf.org`.

[134] George Spanoudakis and Andrea Zisman. Software Traceability: a Roadmap. In *Handbook of Software Engineering and Knowledge Engineering, vol. 3 – Recent Advances*, pages 395–428. World Scientific, 2005.

[135] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, 2009.

[136] The MathWorks. MATLAB/Simulink®. `http://www.mathworks.com/products/simulink/`, 2008.

[137] Walter F. Tichy. RCS - A System for Version Control. *Software - Practice and Experience*, 15(7):637–654, July 1985.

[138] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference Computation of Large Models. In *Proc. of 6th Joint Meeting of ESEC/FSE 2007*, pages 295–304, September 2007.

[139] Julian R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[140] Bert Vanhooff and Yolande Berbers. Supporting Modular Transformation Units with Precise Transformation Traceability Metadata. In *Proc. of the ECMDA Traceability Workshop (ECMDA-TW)*, pages 15–27, November 2005.

[141] Bert Vanhooff, Stefan van Baelen, Wouter Joosen, and Yolande Berbers. Traceability as Input for Model Transformations. In *Proc. of the ECMDA Traceability Workshop (ECMDA-TW)*, pages 37–46, 2007.

[142] Antje von Knethen. Change-Oriented Requirements Traceability: Support for Evolution of Embedded Systems. In *Proc. of the International Conference on Software Maintenance (ICSM'02)*, pages 482–485, 2002.

[143] Antje von Knethen and Barbara Paech. A Survey on Tracing Approaches in Practice and Research. Technical Report Research Report 095.01/E, Fraunhofer IESE, Kaiserslautern, Germany, January 2002.

[144] Jens von Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach, and Yolande Berbers. Constructing and Visualizing Transformation Chains. In *Proc. of the European Conference on Model Driven Architecture (ECMDA)*, pages 17–32. Springer, 2008.

[145] Yuan Wang, David J. DeWitt, and Jin-Yi Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. In *Proc. of the 19th International Conference on Data Engineering*, March 2003.

[146] Sven Wenzel. Scalable Visualization of Model Differences. In *Proc. of the 2008 International Workshop on Comparison and Versioning of Software Models (CVSM'08)*, pages 41–46, May 2008.

[147] Sven Wenzel, Hermann Hutter, and Udo Kelter. Tracing Model Elements. In *Proc. of the 23rd International Conference on Software Maintenance (ICSM'07)*, pages 104–113. IEEE Computer Society, October 2007.

[148] Sven Wenzel and Udo Kelter. Analyzing Model Evolution. In *Proc. of the 30th International Conference on Software Engineering (ICSE'08)*, pages 831–834, 2008.

[149] Stefan Winkler and Jens von Pilgrim. A Survey of Traceability in Requirements Engineering and Model Driven Development. *Software and Systems Modeling*, 9(4):529–565, 2010.

[150] Timo Wolf. *Rationale-Based Unified Software Engineering Model*. PhD thesis, Technische Universität München, Germany, July 2007.

[151] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 1.0. `http://www.w3.org/TR/xslt`, 1999.

[152] World Wide Web Consortium (W3C). XML Path Language (XPath) 2.0. `http://www.w3.org/TR/xpath20/`, 2007.

[153] Zhenchang Xing. Model Comparison with GenericDiff. In *Proc. of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, pages 135–138. ACM Press, September 2010.

[154] Zhenchang Xing and Eleni Stroulia. Analyzing the Evolutionary History of the Logical Design of Object-Oriented Software. *IEEE Transactions on Software Engineering*, 31(10):850–868, October 2005.

[155] Zhenchang Xing and Eleni Stroulia. UMLDiff: an Algorithm for Object-Oriented Design Differencing. In *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 54–65. ACM Press, November 2005.

[156] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stefan Diehl. Mining Version Histories to Guide Software Changes. In *Proc. of the 26th International Conference on Software Engineering (ICSE '05)*, pages 563–572, May 2005.

[157] Xuchang Zou, Raffaella Settimi, and Jane Cleland-Huang. Phrasing in Dynamic Requirements Trace Retrieval. In *Proc. of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, pages 265–272, 2006.

Appendix A

Changes Applied to the Models Used in the Experiments

This appendix given an overview of the kinds of changes that have been applied to the models of the different histories that we used for the controlled experiments in Section 12.1. Each figure shows a histogram chart that visualizes the number changes that have been applied to the models. Each bar represents the changes that have been applied in order to get to the revision of that bar. We differentiate between updates (i.e. changes to attribute values), reference changes, structural changes (i.e. insertions and deletions), and moves. The histograms do also show the size of each revision.

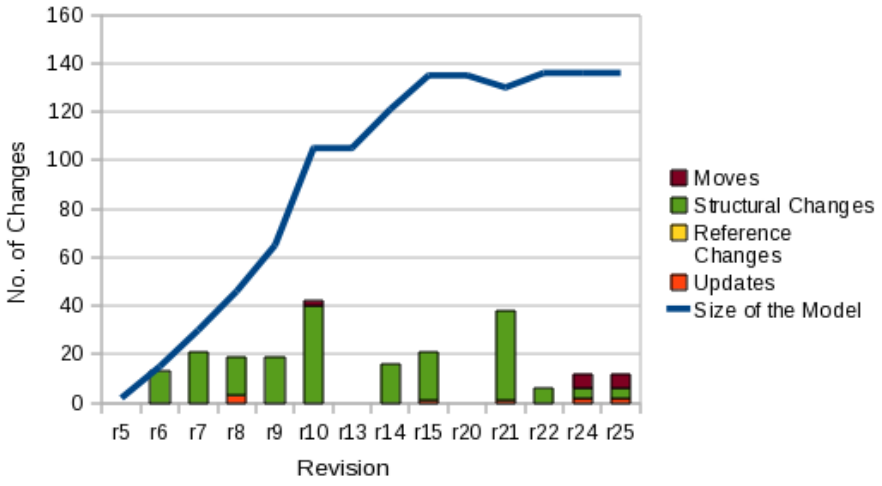


Figure A.1: *Changes applied to the model of history A*

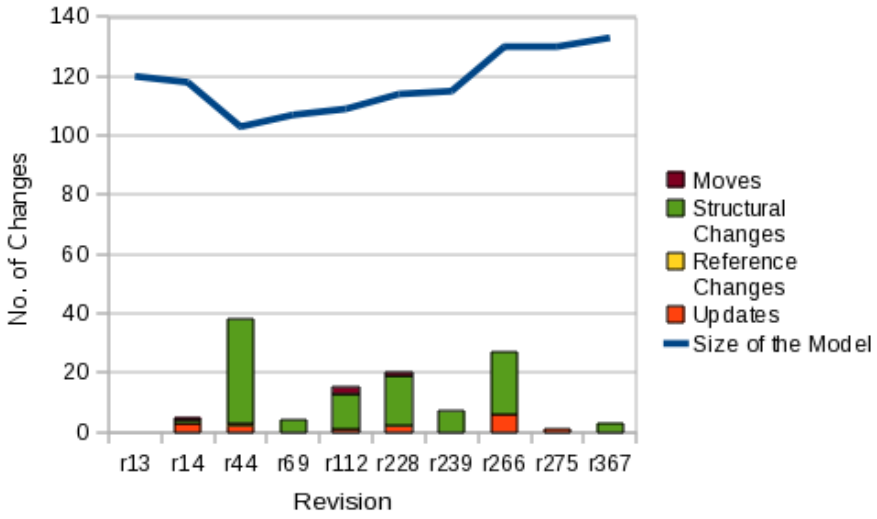


Figure A.2: *Changes applied to the model of history B*

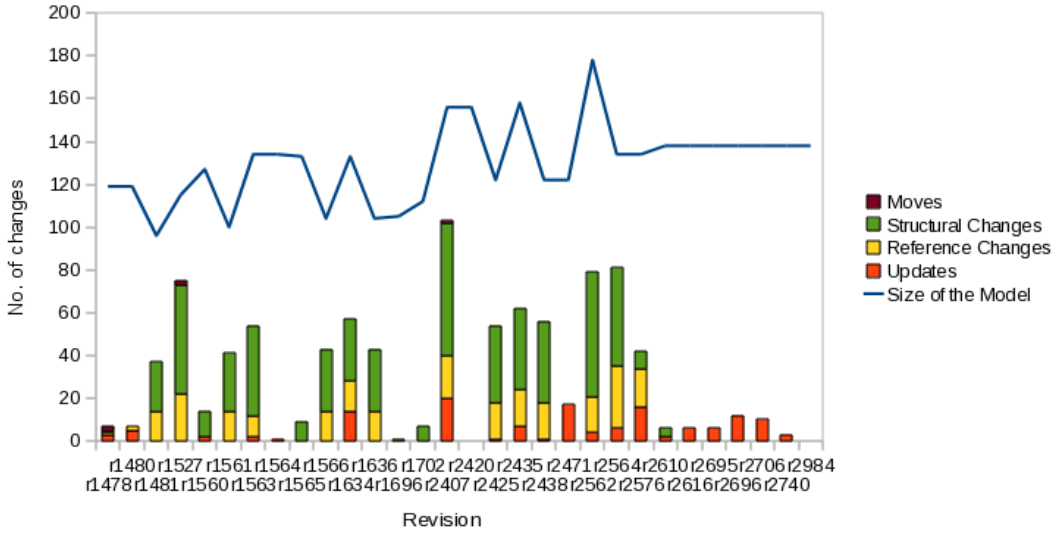


Figure A.3: *Changes applied to the model of history C*

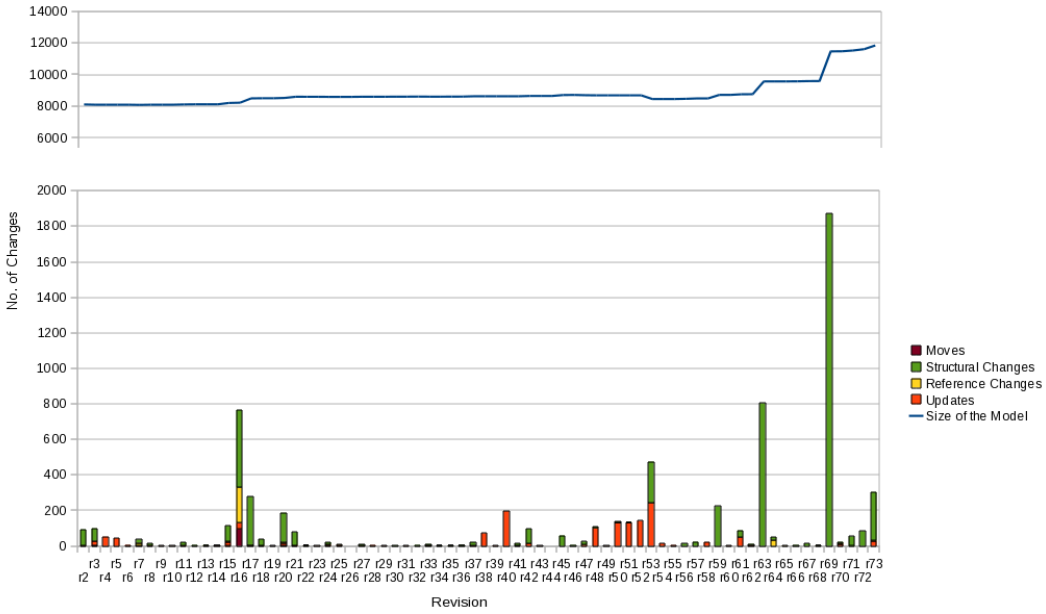


Figure A.4: *Changes applied to the model of history D*

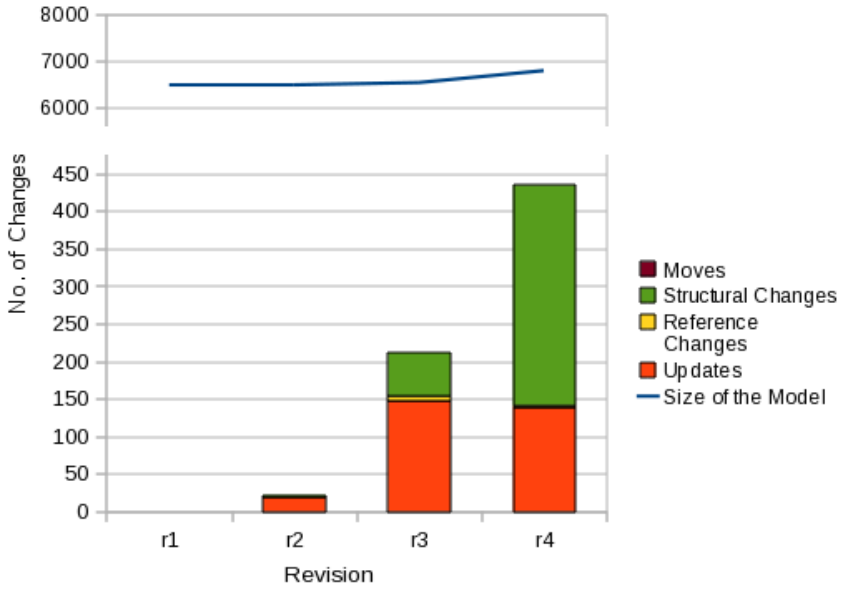


Figure A.5: *Changes applied to the model of history E*

Appendix B

Detailed Results of the Precision-Recall Analysis

The subsequent tables show the results of the precision-recall analyses described in Section 12.1. Each table shows the calculated values for precision and recall for each revision in relation to its successor.

Revision	Precision	Recall
r6	1	1
r7	1	1
r8	1	1
r9	1	1
r10	1	1
r11	1	1
r13	1	1
r18	0.9855	1
r14	1	1
r23	0.9917	1
r15	1	1
r16	1	1
r20	1	1
r17	1	1
r21	1	1
r22	1	1
r24	1	1
r25	1	1

Table B.1: *Results of the PR analysis of history A*

Revision	Precision	Recall
r14	1	1
r44	1	1
r69	1	1
r112	1	1
r228	1	1
r239	1	1
r266	1	1
r275	1	1
r367	1	1

Table B.2: *Results of the PR analysis of history B*

Revision	Precision	Recall
r1478	1	1
r1480	1	1
r1481	1	1
r1527	1	1
r1560	1	1
r1561	1	1
r1563	1	1
r1564	1	1
r1565	1	1
r1566	1	1
r1634	1	1
r1636	1	1
r1696	1	1
r1702	1	1
r2407	0.9936	1
r2420	1	1
r2425	1	1
r2435	1	1
r2438	1	1
r2471	1	1
r2562	1	1
r2564	1	1
r2576	1	1
r2610	1	1
r2616	1	1
r2695	1	1
r2696	1	1
r2706	1	1
r2740	1	1
r2984	1	1

Table B.3: *Results of the PR analysis of history C*

Revision	Precision	Recall	Revision	Precision	Recall
r2	1	1	r38	1	1
r3	0.996	1	r39	1	1
r4	1	1	r40	1	1
r5	1	1	r41	1	1
r6	1	1	r42	1	1
r7	1	1	r43	1	1
r8	1	1	r44	1	1
r9	1	1	r45	1	1
r10	1	1	r46	1	1
r11	1	1	r47	1	1
r12	1	1	r48	1	1
r13	1	1	r49	1	1
r14	0.9993	1	r50	1	1
r15	0.998	1	r51	1	1
r16	1	1	r52	1	1
r17	1	1	r53	1	1
r18	1	1	r54	1	1
r19	1	1	r55	1	1
r20	0.9988	1	r56	1	1
r21	1	1	r57	1	1
r22	1	1	r58	1	1
r23	1	1	r59	1	1
r24	1	1	r60	1	1
r25	1	1	r61	1	1
r26	1	1	r62	1	1
r27	1	1	r63	1	1
r28	1	1	r64	1	1
r29	1	1	r65	1	1
r30	1	1	r66	1	1
r31	1	1	r67	1	1
r32	1	1	r68	1	1
r33	1	1	r69	1	1
r34	1	1	r70	1	1
r35	1	1	r71	1	1
r36	1	1	r72	1	1
r37	1	1	r73	1	1

Table B.4: *Results of the PR analysis of history D*

Revision	Precision	Recall
2	0.9998	1
3	1	1
4	0.9994	0.9994

Table B.5: *Results of the PR analysis of history E*