State-based vs. Operation-based Change Tracking

Maximilian Koegel, Markus Herrmannsdoerfer, Jonas Helming, and Yang Li

Institut für Informatik
Technische Universität München
Boltzmannstrasse 3, 87548 Garching
{koegel, herrmama, helming, liya}@in.tum.de

Abstract. In recent years, models are increasingly used throughout the entire lifecycle in software engineering projects. In effect, the need for managing these models in terms of change tracking and versioning emerged. However, many publications recognized that existing approaches for Version Control (VC) do not work well on graph-like models, and therefore proposed alternative techniques and methods. They can be categorized into two different classes: state-based and operation-based approaches. There are publications that show advantages of operation-based over state-based approaches in selected use cases. However, there are no results available on the advantages of operation-based approaches in the most common use case of a VC system: review and understand change. In this paper, we present and discuss both approaches and their use cases. Moreover, we present a design of an empirical study to compare a state-based with an operation-based approach in the use case of reviewing and understanding change.

1 Introduction

Today, models are an essential artifact throughout the entire lifecycle in software engineering projects. Model-driven development is putting even more emphasis on models, since they are not only an abstraction of the system under development, but the system is (partly) generated from its models. Consequently, models are about to cover the whole development process from requirements over design to deployment, including management of the process itself. With the adoption of model-driven development in industry, the need for managing these models in terms of change tracking and versioning emerged. Version Control (VC) is already in wide-spread use for textual artifacts such as source code.

However, many publications, e.g. [1–7], recognized that existing VC approaches do not work well on models, which essentially are attributed graphs. The traditional VC systems are geared towards supporting textual artifacts such as source code, managing them on a line-oriented level. In contrast, many software engineering artifacts including models are not managed on a line-oriented level, and thus a line-oriented change management is not adequate. For example, adding an association between two classes in a UML class diagram is a structural

change, which is neither line-oriented, nor can be managed in a line-oriented way. A single structural change in the diagram is managed as multiple line changes by traditional VC systems. Nguyen et al. describe this problem as the *impedance mismatch* between the flat textual data models of traditional VC systems, and graph-based software models [1]. Different approaches have been proposed to cope with the shortcomings of existing methods and techniques to better support change tracking and versioning of graph-based models. They can be categorized into two different classes: state-based and change-based approaches [8].

State-based approaches only store states of a model, and thus need to derive differences by comparing two states, e.g. a version and its successor, *after* the changes occurred [8]. This activity is often referred to as diffing. The diffing process can be viewed as a calculation to derive the change post-mortem, and is generally expensive in computation time.

Change-based approaches record the changes, *while* they occur, raising change to a first class concept. There is no need for diffing, since the changes are recorded and stored, and thus do not need to be derived later on. *Operation-based* approaches are a special class of change-based approaches which represent the changes as transformation operations on a state [8]. The recorded operations can be applied to a state to transform it to the successor state.

Several publications exist that show advantages of change-based and in particular operation-based approaches over state-based approaches in use cases such as conflict detection, merging and repository mining [7, 9–13]. However, there are no results available on the advantages of operation-based approaches in the most common use case of a VC system: reviewing and understanding change. We claim that understanding change is the most important use case of a VC system, as it is required for almost any other use case, e.g. commit, update, merge, etc. Therefore, we believe it essential to conduct experiments on how well this use case is supported by the state-based and operation-based approaches.

In this paper, we discuss representatives of the different approaches as well as the advantages and disadvantages of each type of approach in general. To compare them, we present frequent use cases of a VC system, and show how they are supported by the respective approach. Finally, we present the design of an empirical study we intend to conduct to compare a state-based with an operation-based approach for the use case of reviewing and understanding change.

Outline. Section 2 introduces common use cases of a VC system. Sections 3 and 4 introduce the state-based approach and operation-based approach, respectively. Related work is folded into these two sections, thereby making a separate section for related work obsolete. Section 5 presents the design of the empirical study, and Section 6 concludes the paper with a short summary.

2 Use Cases for a Version Control System

A VC system has to fulfill a lot of use cases, many of which do not differ from a state-based to a change-based system. Consider the use case of baselining, in which the user marks a certain approved version (e.g. a release). Since changes

are not at all considered in this use case, both approaches appear identical from a user's point of view. Consequently, we only focus on use cases where a difference between state-based and change-based systems arises. We derived these use cases from well-known tools, such as the Revision Control System (RCS) [14], the Concurrent Versioning System (CVS) [15], and Subversion (SVN) [16], from research tools such as SiDiff [17] and UNICASE [18], and from publications such as [8] and [19]. Figure 1 illustrates the use cases, that we consider important for discussing the differences between state-based and change-based change tracking. For every use case, we provide a short name, and give a description:

Update. The user retrieves changes between her local version and a target version (mostly the current head version) from the repository. These incoming changes can be reviewed by the user, before they are incorporated into the local working copy of the model. If the user accepts the changes and they do not conflict with local changes, the version of the local working copy is set to the target version, and the changes are incorporated into the local copy.

Commit. The user decides to share the changes from her local working copy with the repository. The user reviews the changes before the commit to ensure that only intentional changes are sent to the repository. If the user proceeds, the changes are sent to the repository to create a new version. In case they conflict with other commits that occurred since the last update, the commit is canceled by the VC system and an update must occur first.

Merge. When incoming changes conflict with existing local changes in the update use case, the merge use case is initiated. The goal of the use case is to filter or transform the incoming and/or local changes, until they no longer conflict. The result will be incorporated into the local workspace. The merging process involves manual work in most cases, requiring to review the changes. Merging may also occur if two branches in the repository are synchronized or rejoined, which essentially requires the same steps.

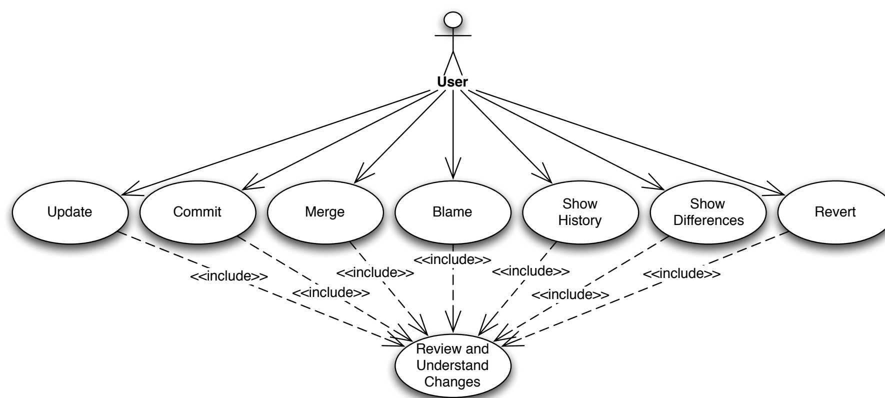

Fig. 1. Use cases of a VC system (UML use case diagram)

Blame. To find out how and by whom a problem or an inconsistency was created, the user is interested in finding recent changes on a certain model part. Typically, the last n changes on a selected set of model elements need to be retrieved. The user reviews these changes to find the causing change.

Show History. The user (often a project manager) reviews the history to get an impression on the current activity in a project. Mostly, she is not interested in individual changes, but in an overview of how many and which type of change on how many artifacts occurred.

Show Differences. The user is interested in reviewing the differences between two versions of a model, for example two releases. The two versions are typically not very close in terms of the number of changes between them.

Revert. The user wants to undo some changes in the local working copy. To ensure that the right changes are undone, she reviews the changes before.

Review and Understand Changes. The user reviews changes to understand what was changed and most importantly how it was changed.

Interestingly, the first seven use cases are placed most prominently in many VC systems and their clients [15, 16, 18]. We claim this is due to the fact that these are the most frequently executed use cases for the majority of users. In all of these seven use cases, the user is reviewing changes in one or another way. As is shown in Figure 1, all use cases thus include the use case "Review and Understand Changes".

3 State-based Change Tracking

State-based approaches derive differences by comparing two states, e.g. a version and its successor, *after* the changes occurred. This activity is often referred to as diffing, and is performed in two phases: matching and comparison. In the matching phase, for each node in one state, the corresponding node in the other state is found. The matching can be based on the similarity of the node's content or on the graph structure it is connected to. If the model supports unique identifiers, the matching can be found in $O(1)$, otherwise $O(n^2)$ are required for n nodes in a model [20, 21]. Chawathe et al. even claim that the matching problem for two states is NP-hard in its full generality [22]. In the comparison phase, each node is compared with its matching partner from the other state to derive changes if there are any. The comparison calculation requires $O(n)$. The space complexity for the whole diffing process is $2n$, since both states need to be present. Change is not a first level concept in a state-based system. The diffing process can be viewed as a calculation to derive the changes post-mortem.

Since the VC system must not be able to observe the changes, while they occur, a total separation of the modeling tools and the VC system is possible. This is a clear advantage over change-based systems. It is even possible to use line-oriented VC system, and to perform diffing on the client side. For example, EMF Compare [23] is a diffing tool for EMF (Eclipse Modeling Framework) [24] models. There are three main disadvantages of the diffing concept: (1) The time

order of changes is lost, and it can not be perfectly derived. For understanding changes, the time order might be important. Moreover, the time order is useful for conflict detection and merging [12]. (2) Groupings of changes to composite changes are lost. Refactoring operations e.g. cause many changes that can be grouped. This reduces the number of changes, and represents the change at a higher level of abstraction. Deriving composite changes, e.g. to detect refactorings, is difficult and in some cases even impossible due to masking problems [25]. (3) The computational complexity for diffing is high, especially if changes between many states need to be derived, or the model is of a large size [20, 21]. We suspect that disadvantages (1) and (2) will reduce the ability of users to understand change, and hope to be able to show this in the empirical study.

Considering the use cases presented in Section 2, we can make the following observations for the state-based approach:

Merge. The merge result can not be as accurate, as composite changes are not available [11, 12]. Refactoring operations for example might only be partly reflected, if not all their caused changes are accepted.

Show History. The computational complexity for diffing could result in a severe performance problem, if looking at many versions and the changes that occurred in between them, since diffing is required for every version.

Review and Understand Changes. Disadvantages (1) and (2) are impacting the ability of humans to understand change. The lost time order of the changes could help to understand the context in which the changes where performed. Composite changes could group many changes that look unrelated, and thus have to be grouped in the user's mental model.

4 Operation-based Change Tracking

In contrast to state-based approaches, change-based approaches record the changes, *while* they occur. This implies that changes in a change-based system are a first class concept. There is no need for diffing, since the changes are already available by design. Operation-based approaches are a special class of change-based approaches which represent the changes as transformation operations on a state [12]. An operation can be applied to a state to transform it to the successor state [8]. Figure 2 shows the simplified taxonomy of operations from the UNICASE system [4, 18]. All operations refer to one ModelElement that is being changed by the operation, and that is unambiguously identified by a unique identifier. An AttributeOperation changes the value of an attribute for a model element. A ReferenceOperation creates or removes one or more links between model elements. A CreateDeleteOperation creates or deletes a model element. A CompositeOperation allows to group several related operations.

The change-based approaches have one disadvantage in common: they require the VC system to be present when the changes occur, i.e. when the modeling tool is manipulating the model. This requires an integration of the VC system into the modeling tool. However, this does not imply that the system must

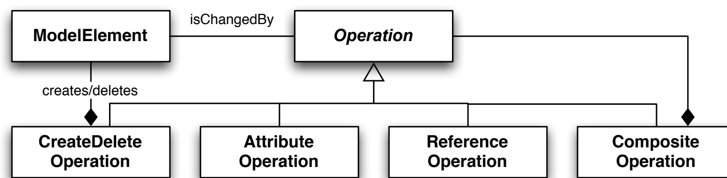

Fig. 2. Taxonomy of operations (UML class diagram)

instrument the modeling tool, but can only use the infrastructure on which the tool is built. For change recording, observer mechanisms can be used, and for composite detection, the command pattern can be used. In case of EMF models, one can rely on the EMF notifications and command stack [24]. This effectively decouples the VC system from the modeling tool. In general, change-based approaches can preserve the time order in which the changes occurred. This is an important information for understanding changes, but is also useful for other applications such as conflict detection and merging [11, 12]. Moreover, the exact times at which the changes occurred can be recorded. Operation-based systems can record composite operations which express the fact that the contained operations occurred in a common context. For example, a refactoring can be captured in a composite operation. This can help to understand changes, but also in conflict detection and merging [10, 11]. Robbes et al. even claim that only an operation-based VC system allows for effective research on evolution, since it provides all the required information [13]. Operation-based systems can provide a method to canonize a sequence of operations, which hides operations that are fully masked by later operations. For example, a Pull up to Superclass refactoring is fully masked by a later deletion of all the involved classes.

Considering the use cases presented in Section 2, we can make the following observations for the operation-based approach:

Update. The operations incoming from the repository are presented to the user. In case the difference between local and target version is large, the system can canonize the operations to get a more compact representation. Conflict detection can fully rely on the operations, their time order and composites to supply a more accurate result and avoid unnecessary conflicts [11]. In general, conflict detectors apply a conservative estimation: If they are unsure about a potential conflict, they raise a conflict to avoid later data corruption.

Commit. The recorded operations can be presented to the user, possibly after a canonization. No diffing is required. Conflict detection can again profit from the additional information.

Merge. The merge can operate on the top level operations. Therefore, less decisions are needed, since many operations are contained in a composite operation. Moreover, the decisions can not partially mask a refactoring as opposed to the state-based case [25].

Show Differences. This use case is best served by a state-based representation. It can be implemented directly by relying on an existing state-based approach or by deriving it from the recorded operations.

Review and Understand Changes. The changes can be shown as operations in the correct time order and grouped as composite operations.

The operation-based approach might seem very different from a state-based approach, but in effect it is only an enhancement. It records additional information that is lost in the state-based approach. In an operation-based approach, we can perform everything that can be done in a state-based approach, by just ignoring the additional information. This boils down to the question whether the additional effort for recording the changes is justified by its advantages. We claim that this is the case, and we hope to find statistical evidence in our empirical study that operation-based change tracking improves the user's ability to review and understand change.

5 Design of the Empirical Study

We conduct the empirical study to answer the following research questions:

1. Do users better understand the changes in a state-based or an operation-based representation? The metrics for understanding are the taken time, self-assessment of users and assessment by interviewers.
2. Which factors influence the user in understanding a state-based or an operation-based representation? Candidate factors are number of changes, kinds of changes and dependencies among changes.

5.1 Setup

We chose representatives for the two different approaches to change tracking.

State-based change tracking is represented by the open-source tool EMF Compare [23] which is the state-of-the-art diffing and merging implementation for EMF. As we do not want to disadvantage EMF Compare by design, the used matching strategy is based on unique identifiers to ensure correct matchings.

Operation-based change tracking is represented by two open-source tools from different domains in order to be more representative: UNICASE for model evolution, and COPE for metamodel evolution.

UNICASE is a CASE tool based on a unified model [18]. The unified model covers the whole development process from requirements over design to deployment, including management of the process itself and its artifacts. UNICASE consists of a modeling tool to author instances of the unified model, and a central repository to persist them including VC. UNICASE is implemented based on EMF, and realizes operation-based change-tracking, conflict detection and merging [4].

COPE is an EMF-based tool to automate the migration of models in response to metamodel adaptation [26, 27]. COPE records the operations carried out on

a metamodel, and allows to attach information on how to migrate models. To significantly reduce effort, COPE allows to reuse generic couples of metamodel adaptation and model migration. The recorded information can be used to automatically migrate models to the newest version of the metamodel.

5.2 Input

Both UNICASE and COPE were used to record operation histories which we use as an input to the empirical study.

UNICASE was employed in a project named DOLLI2 (Distributed Online Logistics and Location Infrastructure 2) at a major European airport. The objective of DOLLI2 was integrating facility management and telemetry data into the tracking and locating infrastructure developed in the previous project, together with expanding the 3D visualization on desktop computers as well as porting it to mobile devices. More than 20 developers worked on the project for about five months. This resulted in a comprehensive project model consisting of about 1000 model elements and a history of over 600 versions.

COPE was employed to reverse engineer the evolution of the metamodels from the Graphical Modeling Framework (GMF) [28]. GMF allows to define a graphical editor by models from which editor code can be automatically generated. The metamodels consist of about 1500 metamodel elements, and the history of over 100 versions.

5.3 Execution

We apply the following systematic process to execute the empirical study:

1. **Choose Users.** We choose 10 users which are familiar with the input, as well as 10 users which are not familiar with the input.
2. **Extract operation-based representation.** For each chosen user, we extract 20 commits from the operation-based histories. Each commit consists of a sequence of operations. To be able to correlate the answers with the complexity of the contained operations, we randomly sample 5 commits for each of the following categories: (1) contains only AttributeOperations, (2) contains also ReferenceOperations, (3) contains also CreateDeleteOperations, and (4) contains also CompositeOperations (for the taxonomy, see Figure 2).
3. **Generate state-based representation.** We generate the state-based representation for all the sample commits using EMF Compare.
4. **Question Users.** We present the commits to the user in a state-based or an operation-based representation. Each user should not receive both representations for the same commit, as the first representation might ease the understanding of the second representation. However, every user should be shown almost the same amount of either operation-based or state-based representation. The user should try to understand the changes. We assess the understanding of the user by means of three metrics: (1) the *taken time*, (2) the *self-assessment* by user, and (3) *assessment by interviewer*. Metric (1)

provides a quantitative answer, whereas metrics (2) and (3) provide qualitative answers. For the qualitative metrics, we use a scale with the following five values: very difficult (=1), difficult (=2), OK (=3), easy (=4), very easy (=5).

5.4 Evaluation

We perform a number of statistical tests to evaluate the measurements. To show the soundness of the measurements, we determine the following correlation:

Self- vs. interviewer assessment. Do the users really understand the changes? The study failed, if both assessments do not vary in a similar way. We carry out a t-test of the null hypothesis, $H_0 : \mu_1 = \mu_2$, on self-assessments and interviewer assessments, where μ_1 and μ_2 are the mean values of the two assessments, respectively.

To answer research question 1, we determine the following correlations:

Self-assessment on state-based vs. operation-based. Is it easier to understand the changes in a state-based or an operation-based representation? We carry out a t-test of the null hypothesis, $H_0 : \mu_1 \leq \mu_2$, on both self-assessments, where μ_1 and μ_2 are the mean values of the two assessments, respectively.

Time on state-based vs. operation-based. Can changes be understood faster in a state-based or an operation-based representation? We conduct a t-test similar to the previous one.

To answer research question 2, we determine the following correlations:

Self-assessment on state-based vs. commit size, self-assessment on operation-based vs. commit size. Is it more difficult to understand the changes, if more operations are involved in a commit? We perform a regression analysis on the assessments, in case the assessments vary in a certain way with increasing commit size.

Self-assessment on state-based vs. operation category, self-assessment on operation-based vs. operation category. Is it more difficult to understand the changes, if the contained operations are more complex? For each $i \in \{1, 2, 3\}$, we perform a t-test of the null hypothesis, $H_0 : \mu_i \geq \mu_{i+1}$, on the assessments of operation category (i) and $(i + 1)$, where μ_i are the mean values of the assessments of the operation category (i).

6 Conclusion and Future Work

We reviewed both state-based and operation-based approaches for change tracking. We compared both approaches by means of typical use cases of VC systems. We are convinced that operation-based approaches are superior to state-based approaches with respect to the use case of reviewing and understanding changes.

To compare both approaches, we will conduct an empirical study as a next step. We expect to observe, that in most cases, the operation-based representation is better suited to understand changes than the state-based representation. Moreover, we want to find out how different factors like e.g. the number of changes influence the activity of understanding changes.

References

1. Nguyen, T.N., Munson, E.V., Boyland, J.T., Thao, C.: An infrastructure for development of object-oriented, multi-level configuration management services. In: ICSE '05, ACM (2005) 215–224
2. Rho, J., Wu, C.: An efficient version model of software diagrams. In: APSEC '98, IEEE (1998) 236–243
3. Kögel, M.: Towards software configuration management for unified models. In: CVSM '08, ACM (2008) 19–24
4. Koegel, M., Helming, J., Seyboth, S.: Operation-based conflict detection and resolution. In: CVSM '09, IEEE (2009) 43–48
5. Bartelt, C.: Consistence preserving model merge in collaborative development processes. In: CVSM '08, New York, NY, USA, ACM (2008) 13–18
6. Ohst, D.: A fine-grained version and confguration model in analysis and design. In: ICSM '02, IEEE (2002) 521
7. Dig, D., Manzoor, K., Johnson, R., Nguyen, T.N.: Refactoring-aware configuration management for object-oriented programs. In: ICSE '07, IEEE (2007) 427–436
8. Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Comput. Surv. **30**(2) (1998) 232–282
9. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: ICSE '08, ACM (2008) 511–520
10. Lie, A., Conradi, R., Didriksen, T.M., Karlsson, E.A.: Change oriented versioning in a software engineering database. SIGSOFT Softw. Eng. Notes **14**(7) (1989) 56–65
11. Mens, T.: A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering **28**(5) (2002) 449–462
12. Lippe, E., van Oosterom, N.: Operation-based merging. In: SDE '92, ACM (1992) 78–87
13. Robbes, R., Lanza, M., Lungu, M.: An approach to software evolution based on semantic change. In: FASE '07. Volume 4422 of LNCS., Springer (2007) 27–41
14. Tichy, W.F.: Design, implementation, and evaluation of a revision control system. In: ICSE '82, IEEE (1982) 58–67
15. Prince, D.: Concurrent Versioning System. `http://www.nongnu.org/cvs`
16. Tigris: Subversion VC System. `http://subversion.tigris.org`
17. Siegen, U.: Sidiff. `http://sidiff.org`
18. J. Helming, M. Koegel: Unicase. `http://unicase.org`
19. Dart, S.: Spectrum of functionality in configuration management systems. Technical report, CMU/SEI (1990)
20. Lindholm, T., Kangasharju, J., Tarkoma, S.: Fast and simple xml tree differencing by sequence alignment. In: DocEng '06, ACM (2006) 75–84
21. Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models. In: ESEC-FSE '07, ACM (2007) 295–304
22. Chawathe, S.S., Garcia-Molina, H.: Meaningful change detection in structured data. In: SIGMOD '97, ACM (1997) 26–37
23. Eclipse: EMF Compare. `http://wiki.eclipse.org/EMF_Compare`
24. Eclipse: Eclipse Modeling Framework. `http://www.eclipse.org/emf`
25. Xing, Z., Stroulia, E.: Refactoring detection based on umldiff change-facts queries. In: WCRE '06, IEEE (2006) 263–274
26. Herrmannsdoerfer, M.: COPE. `http://cope.in.tum.de`
27. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In: ECOOP '09. (2009)
28. Eclipse: Graphical Modeling Framework. `http://www.eclipse.org/modeling/gmf`