Operation-based conflict detection and resolution

Maximilian Koegel, Jonas Helming, Stephan Seyboth
Technische Universität München, Department of Computer Science
Chair for Applied Software Engineering
Boltzmannstrasse 3, D-85748 Garching, Germany
{koegel, helming, seyboth}@in.tum.de

Abstract

Models are in wide-spread use in the software development lifecycle and model-driven development even promotes them from an abstraction of the system to the description the system is generated from. Therefore it is increasingly important to collaborate on models. These models can range from requirements models over UML models to project management models such as schedules. Tool support for collaboration on models is therefore crucial. Traditionally Software Configuration Management (SCM) systems such as RCS [9] or Subversion [10] have supported this task for textual artifacts such as source code on the granularity of files and textual lines. They do not work well for graph-like models with many links since the granularity needed to support them is on the level of model elements and their attributes. For the design of a novel SCM system addressing these requirements it is essential to define how conflicts on models are detected and how they can be resolved. In this paper we present an approach to conflict detection and resolution on models. We employ operation-based change tracking and therefore detect conflicts based on operations. For conflict resolution we propose an integration of SCM with techniques from Rational Management to effectively resolve conflicts.

1. Motivation

With an ever growing number of increasingly large models that are maintained for increased periods of time, SCM systems for models are becoming more and more important. 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. Change pervades the entire software life cycle. Requirements change when developers improve their understanding of the application domain, the system design changes with new technologies and design goals, the detailed design changes with the identification of new solution objects and the implementation changes as faults are discovered and repaired. These changes can affect every work product including models. It is widely recognized that software configuration management (SCM) is crucial for maintaining consistency among, while minimizing the risk and cost of changes to *all* of these artifacts [11].

While many SCM systems exist for source code and other textual artifacts, their support for models with a graph structure is very limited. The traditional SCM systems are geared towards supporting textual artifacts such as source code. Therefore changes are managed on a line-oriented level. In contrast, many software engineering artifacts are not managed on a line-oriented level and therefore a line-oriented change management is not adequate. For example adding an association between two classes in a UML class diagram is neither line-oriented nor can the change be managed in a line-oriented way. A single structural change in the diagram will be managed as multiple line changes by traditional SCM systems. Nguyen et al. describe this problem as an *impedance mismatch* between the flat textual data models of traditional SCM systems and graph-structured software engineering models [7].

We conclude that there is a need to define new methods and techniques for a novel approach to SCM on models. At the core of an SCM system among others the following techniques are required: change tracking, conflict detection, conflict resolution or merging. Change tracking is responsible for finding out what was changed in a local copy of an artifact and to record that change. Conflict detection is concerned with detecting whether simultaneous changes are isolated from each other or not. Changes are conflicting if they result in a different state of the model depending on their serialization. Finally conflicts must be resolved; a common way for resolving conflicts is merging. Merging will discard some of the changes until the remaining changes do not conflict any more.

In this paper we present a novel approach to conflict detection and conflict resolution for an operation-based SCM.

CVSM'09, May 17, 2009, Vancouver, Canada
978-1-4244-3714-6/09/$25.00 © 2009 IEEE

The paper is organized as follows: In section 2 we describe the prerequisites for our approach. In the next two sections we present operation-based change detection and resolution. The paper ends with a short conclusion and of areas of future work we intend to continue our research in.

2. Prerequisites

To evaluate our approach we have implemented it in a tool called unicase [4]. Unicase is a set of editors to manipulate instances of a unified model and a repository to store the model in and to collaborate on the model. Unicase is based on Eclipse technology, in particular the Eclipse Modeling Framework (EMF) and the Graphical Modeling Framework (GMF). Unicase consists of two major components, a rich-desktop client and a server. The client is built on top of Eclipse and deployed as a set of Eclipse plugins or a standalone Eclipse distribution. The client contains editors to manipulate the model in text oriented editor, in graphical editors for diagrams or tabular editors for lists and tables. The client contains a workspace subsystem which automatically records changes to the model. Upon user request, e.g. an update or commit request, the client sends changes to and receives changes from the server. The server (EmfStore) maintains the history of the project as a version tree and provides persistency and access control. The version model is a tree of versions with revision and variant links [5]. Unicase is open source and is available for download on its project home page [4].

In the following subsections we describe the subsystems and aspects of unicase that are important for this paper. The first subsection describes the model we use to represent the system under development and all project related data. In the second subsection we explain Ecore the meta model of our model. Our approach to operation-based change tracking is presented in the third subsection.

2.1 Unified Model

Our model is based on the concept of a unified model [12]. In this section we will briefly define and describe this term. A model is an abstraction of system. In Software Engineering a model can describe the system under development on different levels of abstraction, at different points in time and from different points of view. Typically in a software development life-cycle many models of the system under development are produced. These models range from requirements models to class models of the entities involved in the system. However models in a software development project may also build an abstraction of the project itself. Many management artifacts fall into this category, such as organizational charts, schedules or work breakdown structures. Models typically define model elements and model links. Model elements are nodes consisting of a certain number of attributes of various type. For example in a UML class model there are class nodes including but not limited to the attributes name of type String and isAbstract of type boolean. Model links define the connections that are valid among nodes. In the UML class model example, links include association, aggregation and composition links. Essentially all instances of models are graphs with instances of model elements as nodes and instances of links as edges. Since several models can describe the same system under development, it is often useful to add additional links connecting these models. For example, it can be essential to link a requirement from a requirement model with a use case from a use case model to express that the use case is describing the requirement in more detail. We therefore distinguish two types of links: intra-model links and inter-model links. Intra-model links connect instances of the same model, such as an association link in a class model. Inter-model links connect instances from different models such as a requirement-detailed-in-use-case link. A unified model defines models with their model elements and intra-model links and last but not least inter-model links. Tracking change and managing revisions on an instance of a unified model is essentially change tracking and versioning on a graph.

2.2 Ecore model

Since our change tracking is generally independent from a certain model, it is important to understand the metamodel of our model. Since we use the Eclipse Modeling Framework (EMF) [2] for modelling our model, our metamodel is Ecore. In this section we will shortly describe the Ecore model. Figure 1 shows a UML class diagram of the Ecore model. An instance of that model consists of a number of *Packages*, each package can contain classes. A *Class* has a unique name within a package and consists of a number of features. Note that feature is *not* a term from Requirements Engineering in this context. A *Feature* is similar to an attribute in UML and can either be a ReferenceFeature or an *Attribute*. An *Attribute* is an attribute of a simple type, such as string, integer, boolean or enumeration and it can not refer to an instance of a class. In contrast a *ReferenceFeature* can only refer to the instance of a class. A feature in general has a name which uniquely identifies the feature within a class and a multiplicity, that implies how many instances of the respective type can be hold in the feature. For a string-type attribute multiplicity (n,m) means that the feature is a list or set of a minimum of n and a maximum of m strings. The value -1 denotes infinity for multiplicity boundaries. A reference feature can be bidirectional. This means that an instance B of a class can only be contained in

the reference feature of an instance A if B in turn contains A in one of its reference features. For this purpose the model shows a reflexive association for ReferenceFeature. An instance of a ReferenceFeature can define that it is bidirectional to another instance of a ReferenceFeature. EMF will automatically take care that bidirectional associations are always in sync. In other words EMF will automatically add B to the respective feature when A is added to the opposite feature in B. Reference features also support the boolean attribute *containment*. A reference feature where this attribute is set to true is also called containment reference feature or containment feature for short. A containment feature adds another constraint to the elements that are contained in this feature: Every element can only be contained in exactly one containment feature. So every element has no or exactly one container.

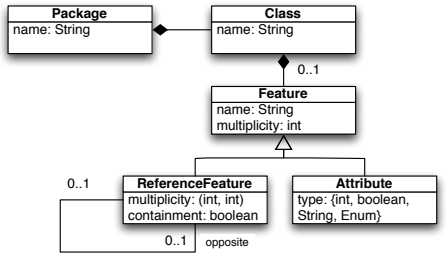

Figure 1. EMF Ecore (UML class diagram)

2.3 Operation-based change tracking

Since the unified model is highly extensible the change tracking is based on the meta-model (Ecore). So new model element classes or changes to the model can be facilitated without changing the SCM. The basic idea of operation-based change tracking is that it is changed-based not state-based. In a state-based SCM the changes are derived by diffing two states of a model instance. The diffing process for state-based SCM systems is twofold: First the two states are matched, finding out which node in the one state is related to which node in the other if any. Second the changes are derived by looking at the nodes' attributes. There are two main disadvantages of this approach: First matching and diffing complexity depend on model size. While both can be reduced to O(n) where n is the size of the model by introducing unique identifiers, this is still considerable. Secondly, changes can not be discovered at the same level of detail as with change-based approaches. Only the result of an operation can be observed not its intermediate states. Consider a state diagram where state A, B and C are connected in sequence at first while later only state A is connected with C and B is isolated. Two essentially different operations can lead to this result, either A changed its successor to C or C changed its predecessor to A. To understand the intention of a change this might be crucial information. Another good example are refactoring operations where the user triggers one operation resulting in many changes to the model. For a state-based approach it is impossible to know which changes are part of a refactoring. In change-based SCM systems the changes are not derived but directly recorded while they occur. This is difficult if the SCM cannot control the editors, which probably is why they are not yet in wide-spread use. While for source code the programming language provides a common standard and different editors can work on the same resources, this is not true for most models; they are already coupled with an editor. Furthermore the recording can be coupled with the model and not its editors further reducing the implications of this coupling. A change-based SCM works much like the command pattern applied in many implementations of undo/redo stacks. The basic idea of change-based SCM has been presented in [6]. Each time a change is triggered by some user interaction the operation that was performed on the model is recorded and stored.

Figure 2 shows the taxonomy of our operations. We will discuss the taxonomy top-down in the following. An *AbstractOperation* is a change with a globally unique id that refers to exactly one *ModelElementId*. A *ModelElementId* is a globally unique identifier for a model element. *ModelElement* is the superclass of all model elements defined in the respective model. A project in our terminology is an aggregation of model elements that build an abstraction of the system under development including its project management artifacts. An *AbstractOperation* provides two methods: apply and reverse. The apply method executes the change that was recorded by the operation on a project and its model elements. This is important for sending changes to different nodes to replicate the changes from another node. The reverse method builds an instance of AbstractOperation that we call reverse operation. Reverse operations implement the following contract: The resulting project after sequentially applying an operation and its reversed operation is identical with the input project. The reverse method is needed for local undo or revert operations and for operation-based merging as presented in [5]. All subclasses of AbstractOperation are self-contained; that is they do not have direct associations to the model instance they originate from but refer to model elements by their model element id. This is an important property since the changes need to be transported among different nodes without the model instance itself. Also revert could not work without additional input otherwise.

There is one exception to the reference by id rule: *CreateDeleteOperation*. It describes the creation or deletion of a model element depending on its isDelete attribute. Since a model element that is created has not been seen by other

nodes and can therefore not be referenced by id, it needs to be contained in the operation. Whenever a model element is created, change tracking will copy the element and create an instance of CreateDeleteOperation that contains the copy.

CompositeOperation builds a composite pattern with AbstractOperation and its other subclasses. A CompositeOperation contains a method cannonize to minimize the number of operations contained in it. For example if two operations update the same attribute of the same model element, they can be folded into one operation. A CompositeOperation is used to aggregate operations that result from a refactoring or automatic generation such as batch renaming or automatically deriving a system test case from a use case.

FeatureOperation is an operation on exactly one feature of a model element; it contains the feature name. Its subclass *AttributeOperation* describes a change to a value of a simple attribute of a model element. It contains the old and new value of a simple attribute such as boolean, integer or string. A *MultiAttributeOperation* describes a change to a list- or set-valued attribute. Therefore it can add or remove values from the list or set denoted by the isAdd attribute at a given index. A *ReferenceOperation* records a change to a reference feature. Its subclases *SingleReferenceOperation* and *MultiReferenceOperation* only differ in the multiplicity of the feature. *MultiReferenceOperation* records a change on a reference feature with a multiplicity upper bound greater than one. Finally there are move operations for ordered multi attribute and multi reference features. *MAttributeMoveOperation* and *MReferenceMoveOperation* respectively contain the old and new index and the values to be moved.

By design all operations also hold the necessary information to create their reverse operation. For example for a MultiReferenceOperation we just need to change the boolean attribute isAdd to reverse it and for an AttributeOperation we swap old and new value.

3 Operation-based conflict detection

Conflict detection is the activity of determining whether two sets of changes are interfering with each other. As a general definition we can say two operations are conflicting if the result of applying them to a project depends on the actual serialization [6]. We can easily expand this definition for sets or lists of operations: Two lists of operations are conflicting if any operation in the first list conflicts with any operation in the second list. Since every conflict will result in a manual merging operation, it is crucial to detect conflicts as fine-grained as possible and minimize the number of detected conflicts while not losing information (lost update) or violating model integrity. On the other hand for performance we can not just apply all pairs of operations in both serializations to detect conflicts, but only calculate an approximation to this. We detect conflicts on the attribute level of model elements. In general we flag two operations as conflicting if they change the same values in the same attribute of the same model element to a different result. If they do, applying both will result in a different model instance as the model will differ and at least in this attribute value.

For later conflict resolution and visualization conflict detection must support an method to determine all operations that are involved in a conflict. Since there might be operations that require a certain operation to be applied we can not simply discard operations in a merge. We need two relations *conflicts* and *requires*. The *conflicts* relation defines when two operations are conflicting according to our definition of a conflict. The second relation determines whether one operation can *not* be applied without the other. For example an attribute operation on element A cannot be applied without a previous create operation for the element A. In general an operation A requires an operation B if A is not applicable on a project without B. An operation is not applicable if the operation can not be performed since it refers to model elements or attribute values that do not exists. The requires relation is transitive. These two relations can define a conflict detection strategy in a strategy pattern that is used by a conflict detector to calculate transitive closures on the required relation, detect conflicts in lists of operations or assists in merging. Note that there are some conflicts that can be resolved automatically. For example adding elements to a reference feature can be a conflict since it depends which elements are added first. Nevertheless the system can automatically merge the two changes and just warn the user about the auto resolved conflict. In the following we describe the conflict detection strategy we use in our approach by defining the *requires* and *conflicting* relations:

An instance of CreateDeleteOperation with isDelete set to false is *required* by any operation that refers to that model element. It can refer to the model element either by directly changing one of its attributes or an opposite attribute in a bidirectional attribute or lastly by containing the elements identifier as a value in a reference operation. A composite operation is required by another operation if any of its contained operations is required by the other operation. A MultiAttributeOperation or a MultiReferenceOperation with attribute isAdd set to true is required by any later operations of the same type but with isAdd set to false, if the value attributes of both have a non-empty intersection.

An instance of CreateDeleteOperation with isDelete set to true is *conflicting* with any operation that refers to that model element. It can refer to the model element either by directly changing one of its attributes or an opposite attribute in a bidirectional attribute or lastly by containing the elements identifier as a value in a reference operation. A

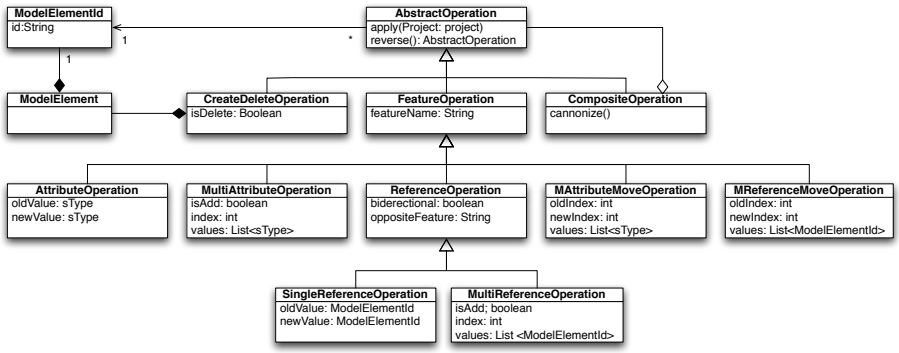

Figure 2. Operations Taxonomy (UML class diagram)

composite operation is conflicting with another operation if any of its contained operations are conflicting with the other operation. Two instances of MAttributeMoveOperation or MReferenceMoveOperation conflict if their values are not disjoint and the target index is different. Two SingleReferenceOperations or AttributeOperations are conflicting if they refer to the same model element and the same feature and have a different new value. There are a number of conflicts that can be resolved automatically, i.e.: Two reference operations or multireference move operations can be conflicting if they refer to the same feature of the same model element. However the results for the different serializations will only differ in the order of the elements in the reference feature, so it can be auto resolved by choosing either one.

This fine-grained conflict detection strategy results in few conflicts only and many of them are automatically merged. In our experience there are far less CreateDeleteOperations than other operations. Since they account for most unresolvable conflicts there are only few of these conflicts in comparison to the total number of operations.

4 Operation-based conflict resolution

For all detected conflicts that can not be resolved automatically, user interaction is required to choose from the conflicting operations. From an abstract point a conflict that cannot be resolved automatically is not a technical problem but a representation of a conflicting opinions on a model from two or more project participants. In a traditional merge process the user that is committing last can and must decide about the conflicts. Before resolving all conflicts he or she cannot proceed. To facilitate not only solving the technical problem of a conflict but the real conflict of opinions in a development team we propose to use well-known concepts from Rationale Management. A conflict represents different opinions in a development team about a certain part of the model and should therefore not always be solved by only one of the members involved in the conflict on his or her own but by all involved parties in a collaborative effort.

Rationale Management Models such as QOC [1] or IBIS [8] work with issues that describe a question or problem that needs to be solved. The unicase rationale model as part of the unified model is based on these concepts and is denoted in Figure 3 as classes filled in white. An *Issue* is a question that needs to be solved and consists of *Proposals* for solving the issue and criteria that can be used to assess or discuss the different proposal. For discussing a proposal there are arguments that can be based on criteria to support or argue against a proposal. A proposal can be designated to be the current solution of the issue by the *isSolved* association. An issue is *facilitated* by an *OrganizationalUnit* such as a user and many units can participate in solving the issue. Also an issue can be annotated to model elements the problem is related to.

For conflict resolution we implemented operation-based merging as presented in [5]. We added *issue-based merging* to improve conflict resolution on a non-technical level. All classes in grey have been added to support issue-based merging, see Figure 3. Since a conflict is a problem that needs to be solved we introduce *MergingIssue* as a subclass to issue. A MergingIssue has a baseVersion that is the version the conflict was detected in and a target version where the issue was created. *MergingProposal* is a subclass of proposal with the additional attributes source and target version. A merging proposal is a proposal to accept a certain operation. A merging issue has normally two proposals for the two conflicting operations from a merge. It is possible however that even more users participate in a conflict, this will increase the number merging proposals. The proposal holds a reference to the operation it represents and to the versions the operation occurred in between. The creation of an issue for a conflict is an additional dimension of choice in the merge process: The user can either accept one or the other operation or ignore the conflict. Additionally the user

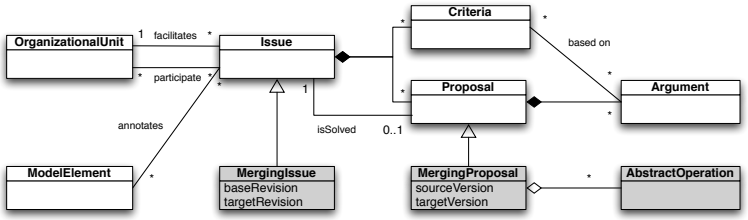

Figure 3. Merging Issue Model (UML class diagram)

can select to create an issue. If the user selects to ignore a conflict an issue is created automatically. The issue is assigned to the user performing the merge as facilitator and the other user(s) involved in the conflict as participant(s). For the conflicting operations proposals are created that refer to the respective operations. The issue is also annotated to the model elements the conflict operations refer to since every operation refers to a model element id. Finally if the user accepted one of the two operations but still decided to create an issue to document his or her choice, the respective proposal is designated as solution to the issue. In contrast to normal issues, merging issues support the execution of a proposal. If a proposal is selected as solution, the operation of the proposal is applied on the project. Thereby the proposal is implemented. If a proposal is removed as solution, the operation is reversed and applied on the model and therefore is undone. If a conflict occurs in the execution of a proposal, the operation that causes the conflict is added as additional proposal and designated as solution. By allowing conflict resolution to be a collaborative, users are not forced to resolve conflicts immediately without understanding the full impact of their resolution. Since the issues and their proposals are created automatically and linked with all impacted resources there is only little overhead.

5 Conclusion and future work

We have implemented operation-based change tracking and conflict detection in the unicase tool to evaluate our approach. At the moment we are extending the implementation with the issue-based merging as presented in Section 4. The tool including the operation-based SCM is currently evaluated in a project with 20 developers. Preliminary results show that the approach is feasible, however there is not much support for building such a system in terms of previous experience and frameworks. We hope to get the issue-based merging into use before the end of the project to be able to compare the results with and without the possibility to create merging issues. We are already convinced that operation-based change-tracking and conflict resolution are superior to state-based approaches in terms of the quality of the changes and in terms of computational complexity.

References

[1] V. Bellotti, A. MacLean, and T. Moran. What makes a good design question? *SIGCHI Bull.*, 23(4):80–81, 1991.

[2] Eclipse.org. Eclipse modeling framework. http://www.eclipse.org/modeling/emf/.

[3] Eclipse.org. Emf compare. `http://wiki.eclipse.org/index.php/EMF_Compare`, 2009.

[4] J. Helming, M. Koegel. unicase project home. `http://unicase.org`, 2009.

[5] M. Koegel. Towards software configuration management for unified models. In *ICSE '08, CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models*, pages 19–24, New York, NY, USA, 2008. Technische Universität München, ACM.

[6] E. Lippe and N. van Oosterom. Operation-based merging. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 78–87, New York, NY, USA, 1992. ACM Press.

[7] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 215–224, New York, NY, USA, 2005. ACM Press.

[8] H. Rittel and W. Kunz. Issues as elements of information systems. Technical report, Institut fur Grundlagen der Planung, University of Stuttgart, 1970.

[9] W. F. Tichy. Design, implementation, and evaluation of a revision control system. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 58–67, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[10] Tigris.org. Subversion version control system. `http://subversion.tigris.org`, 2009.

[11] T. View. IEEE Standard for Software Configuration Management Plans. *IEEE Std 828-2005 (Revision of IEEE Std 828-1998)*, pages 1–19, 2005.

[12] T. Wolf. *Rationale-based Unified Software Engineering Model*. Dissertation, Technische Universität München, July 2007.