Selective Backtracking of Model Changes

Iris Groher and Alexander Egyed
Johannes Kepler University
Institute for Systems Engineering and Automation (SEA)
{iris.groher, alexander.egyed}@jku.at

Abstract

Backtracking is necessary when design alternatives are explored or dead ends are reached. Unfortunately, current approaches support chronological backtracking only (undo or version control), where the designer is forced to undo intermittent changes even if they are not related to what should be backtracked. This work introduces an approach for selective backtracking during software modeling where previously discarded design changes are recovered without having to undo intermittent changes. Selective backtracking is a challenge because during multi-view modeling, we must understand how changes across multiple views are connected – in order to undo them together and thus avoid undesired inconsistencies. Our approach automatically discovers dependencies among design changes and is thus able to guide the designer during selective backtracking.

1. Introduction

Software modeling emphasizes separation of concerns [1] through different views (e.g. structural, behavioral, scenarios) [2]. While each view is described separately, these views are dependent on one another in that changes to one view can affect others. Figure 1 shows a simplified video-on-demand system and two of its views: a class and a sequence diagram. Both views are captured and understood separately. However, the messages in the sequence diagram should refer to methods declared in the class diagram. Such dependencies among views are typically explored through the help of consistency rules [3]. Rules that express dependencies among views are typically reusable across different application domains.

The downside of most modeling languages is that they are designed to define solutions only and do not have constructs for remembering the changes made along the way (the design history [4]). However, changes matter [5] and designers often desire to backtrack the design: for example when dead ends are reached, multiple design alternatives are explored (i.e., with the implicit understanding of backtracking to undo the alternative if it was not satisfactory), or simply to recover something that was previously discarded. Remembering changes is also important for understanding design decisions that have led to a specific solution.

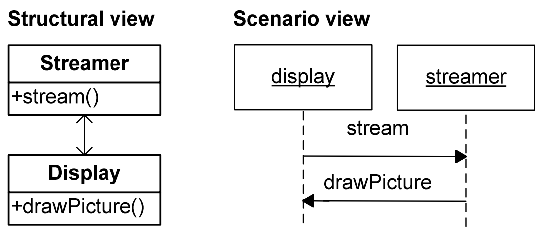

Figure 1. Dependencies among Views

Selective backtracking allows designers to undo specific changes in the past without necessarily having to undo the many intermittent changes made thereafter. Selective backtracking almost always affects multiple model elements which need to be backtracked as a logical group to minimize inconsistencies. For example, backtracking the class diagram to a version where the method *stream()* did not exist also requires the backtracking of the sequence diagram to a version where the message stream did not exist – or else the backtracking would cause an inconsistency.

There are a range of existing solutions on how to deal with backtracking. Most modeling tools provide undo mechanisms but they force designers to undo changes chronologically. Versioning mechanisms (CVS or Subversion) do not solve this problem either. Versioning mechanisms do allow parts of the model, even individual model elements, to be versioned separately. However, the real complexity of selective backtracking is in understanding how the undoing of one model element affects others.

Our work provides a technique for selective backtracking that automatically discovers dependencies among design changes to guide the designer. The designer chooses what model element(s) and versions to backtrack. The technique then informs the designer of other model elements that should be backtracked also to minimize inconsistencies caused.

2. Problem and Illustrative Example

Figure 2 illustrates selective backtracking on two versions of the video-on-demand system. Version 1 represents an early design snapshot (it is identical with Figure 1). Version 2 represents a later design snapshot. We see that the designer made several changes: (1) the *streamer* no longer calls *drawPicture()* on the *display* and the *display* now pulls the pictures to draw from the *streamer* (*getPicture*), (2) there is thus no longer a need for a bi-directional relationship between the *Streamer* and the *Display* class, and (3) a *connect()* method was added to class *Streamer* including a corresponding message in the sequence diagram.

Imagine that the pulling mechanism for getting pictures (*display* calling *getPicture()*) is no longer desired. Instead, the designer desires to recover the discarded pushing mechanism used in version 1 (*streamer* calling *drawPicture()*). The designer thus desires to backtrack the design history to eliminate *getPicture* and replace it with *drawPicture*. If the designer does not want to loose all intermittent changes such as the addition of *connect()* in *Streamer* and its corresponding message in the sequence diagram then a chronological backtracking is not desirable. Neither may it be desirable to manually “re-discover” what it takes to add the missing message in the sequence diagram. Indeed, the simple replacement of the *getPicture* message with the *drawPicture* message causes two inconsistencies because there is no *drawPicture()* method in *Display* and the message call from *streamer* to *display* would violate the uni-directional calling relationship between their classes.

Inconsistencies caused during backtracking thus help us discover logical dependencies among model elements. This is intuitive since consistency rules describe conditions that a model must satisfy for it to be considered a valid model. Table 1 describes the two consistency rules used above more formally.

Table 1. Sample Consistency Rules

Rule 1	**Name of message must be declared in method**
	operations=message.receiver.base.methods
	return(methods->name->contains(message.name))

Rule 2	**Calling direction of object must match class**
	in=object.base.incomingAssociations
	out=object.incomingMessages->sender.base.
	outgoingAssociations
	return (in.intersectedWith(out)<>{})

Consistency rule 1 states that the name of a message must match an operation in the receiver’s class. If this rule is evaluated on message *stream* in version 2 of the sequence diagram then it first computes all methods of the message’s receiver class. The receiver of the *stream* message is the object *streamer* of type (i.e, base in UML) *Streamer* and the class’ methods are *stream(), connect(), and getPicture()*. The consistency rule is satisfied (i.e., consistent) because the set of method names in *Streamer* contains the one with the name *stream* – the name of the message. Consistency rule 2 then validates whether the calling direction indicated in the class diagram matches the calling direction of the messages in the sequence diagram.

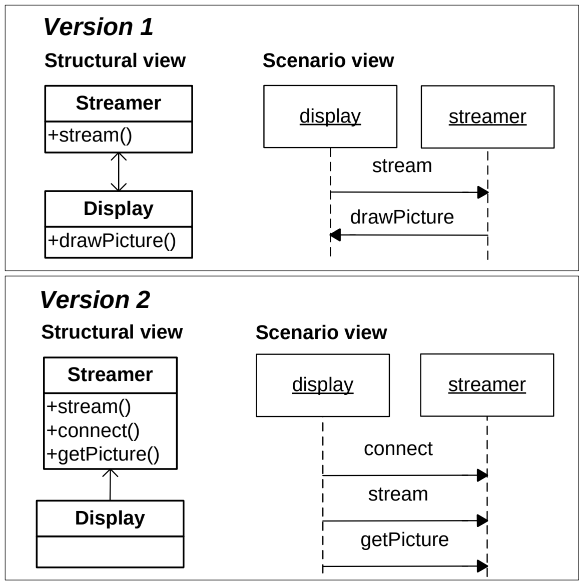

Figure 2. Two Versions of a Video on Demand System

3. Selective Backtracking Approach

Our approach monitors the designer and records design changes in a design history. The design history maintains the changes of each model element and field separately – thus allowing a designer to select arbitrary versions of model elements for backtracking. We presume that the designer initiates the backtracking by choosing what version(s) for what model element(s) to recover. For example, the designer way want to replace the existing *getPicture()* message with a now-deleted *drawPicture()* message in the sequence diagram. This implies two designer-invoked changes: the deletion of *getPicture* and the re-creation of the previously deleted *drawPicture* message in its place.

The approach thus automatically creates, modifies, or deletes the model element(s) selected for backtracking according to Table 2. If the model element was deleted but should be recovered then the element must be re-created and modified to reflect the desired state (version) – as with the message *drawPicture*. If the element still exists (was not deleted) but should be eliminated then the element must be deleted – as with *getPicture*. Or, if an earlier state of an existing element should be recovered then the element must be modified.

Table 2. Backtracking Effects on Model Elements

	eliminate element	recover earlier element state
element still exists	delete	modify
element was deleted	-	create+modify

This simple algorithm, however, does not cover the problematic situation where the deletion, creation, or modification of model elements causes inconsistencies. Inconsistencies during backtracking reveal logical dependencies that must be addressed. For this purpose, this paper uses the UML/Analyzer tool [3] for the instant consistency checking of design models. The tool helps designers in detecting and tracking inconsistencies and it does so correctly and quickly with every design change (i.e., a backtracking step is essentially a design change). What is novel about the UML/Analyzer approach is that it treats every evaluation of a consistency rule separately. We speak of constraint instances. For example, consistency rule 1 in Table 1 must be evaluated three times in version 2 of Figure 2 – once for every message. The UML/Analyzer approach thus maintains three constraint instances: we call them *C1_connect*, *C1_stream*, and *C1_getPicture*. All three constraint instances are evaluated separately as they may differ in their findings (although all are currently consistent).

Backtracking affects model elements which, in turn, affect the consistency of constraint instances. The changes caused during backtracking thus trigger re-evaluations of constraint instances. For this purpose, the UML/Analyzer approach automatically maintains a change impact scope that reveals how model changes trigger re-evaluations of constraint instances. With the creation and deletion of model elements, constraint instances must also be instantiated or disposed. This is also already implemented. The details of the approach are discussed in [3, 6] and omitted here.

When backtracking the *drawPicture* message (to a version in which it existed) and the *getPicture* message (to a version in which it did not exist), the UML/Analyzer approach automatically recognizes that the constraint instance *C1_getPicture* becomes obsolete and a constraint instance *C1_drawPicture* must be instantiated. This new constraint instance is inconsistent because there is no method in the current version 2 of the class diagram with a matching name (no such method was yet recovered from version 1). In addition, the UML/Analyzer approach recognizes that an existing constraint instance, namely C2_display, must be re-evaluated. C2_display is an instantiation for consistency rule 2 on the sequence object *display*. Its task is to ensure that messages arriving at *display* do not violate the calling relationship imposed in the class diagram. This constraint instance was previously consistent (all messages where invocations from *display* onto *streamer* as allowed in the class diagram); however, with the recovery of the *drawPicture* message, this constraint instance becomes inconsistent because *drawPicture* is an invocation from *streamer* onto *display* which is not allowed with the uni-directional relationship in the class diagram.

Table 3. Backtracking Effects on Constraint Instances

constraint instances		after		
		consistent	inconsistent	disposed
before	consistent	no problem	problem	no problem
	inconsistent	no problem	no problem	no problem
	disposed	no problem	problem	no problem

Our approach reacts to inconsistencies caused during backtracking. The goal is simply to avoid them. Table 3 reveals that a constraint instance is problematic if it was consistent before backtracking but no longer is after backtracking (as with *C2_display*); or if the backtracking causes the instantiation of a constraint instance that is then inconsistent (as with *C1_drawPicture*). Both cases imply that the backtracking was incomplete and other model elements must be changed also.

In order to narrow down the search for these other model elements, our approach investigates the change impact scopes of the problematic inconsistencies (we mentioned above that this scope is automatically computed to understand which constraint instances to re-evaluate if a model element changes). Our finding is that in order to fix the inconsistencies caused by backtracking we simply have to backtrack some of the model elements in their respective change impact scopes until a consistent state is found (if any). Fortunately, the change impact scope is conservative and thus contains all model elements needed for further backtracking. Moreover, empirical studies have shown that the scope stays small [3].

The change impact scope is determined by observing the run-time behavior of consistency rules during their evaluation. To this end, the UML/Analyzer approach incorporates the equivalent of a model profiler for consistency checking which lists all model elements accessed during evaluation.

For example, the evaluation of the problematic constraint instance *C1_drawPicture* accesses the message *drawPicture* first, then the message's receiver object *streamer*, its base class *Streamer*, and finally the methods *stream()*, *connect()*, and *getPicture()* (recall earlier). The change impact scope of *C1_drawPicture* is thus {*drawPicture*, *streamer*, *Streamer*, *stream()*, *connect()*, *getPicture()*}.

Our approach investigates each problematic inconsistency. For each inconsistency, it locates the change impact scope elements and identifies which combinations of model element versions have existed during the lifespan of the model element being backtracked. For example, for *C2_display* we find that class *Display* was different in version 1 (note: class *Streamer* was also different but it was not in the change impact scope of *C2_display*). To investigate whether the older version of *Display* resolves the inconsistency, we simply recover its version and re-evaluate all affected constraint instances. Since the older version of *Display* is consistent, we suggest it to the designer. Each such exploration of an inconsistency thus identifies zero, one, or more combinations on how to resolve the inconsistency (zero implies no solution).

The backtracking algorithm is in fact much more sophisticated than can be described in this short paper. We have a built-in notion of minimal backtracking. Also, while the backtracking is explored for each inconsistency separately, there must be an overlapping notion of version correctness: i.e., the backtracked version can only settle on a single version for each model element and it is illegal to resolve two inconsistencies with different versions of the same model element. Furthermore, collections must be dealt with differently during backtracking than singular values. And, not all model elements in a change impact scope must be changed during backtracking. These details will be discussed in future publications.

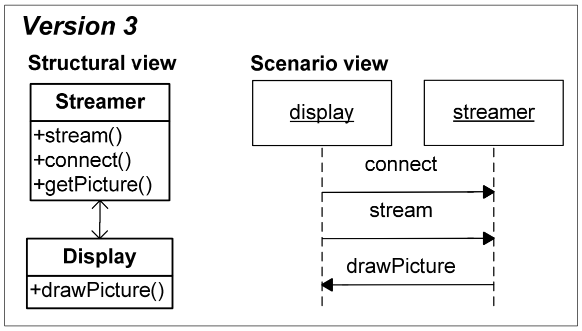

Figure 3. Automatically Suggested Backtracking Solution

Figure 3 depicts the result of the backtracking of *getPicture* and *drawPicture*. The backtracked version contains the recovered message *drawPicture*, its method, and the bi-directional relationship – without losing intermittent changes such as message *connect*.

4. Conclusion

This paper discussed an approach for selective backtracking of design changes. As we have seen, selective backtracking is a difficult problem because of the complex, logical dependencies among design changes. We solved this problem by automatically discovering dependencies via the UML/Analyzer consistency checking approach – where inconsistencies and their fixes reveal the dependencies in question. The approach was evaluated on smaller samples thus far. The next steps are a more comprehensive evaluation in context of several larger models (based on models we used in [3]) and the empirical analysis of the scalability factors. We believe that selective backtracking is vital to software engineering. Current solutions are highly inadequate – mainly because they fail to reveal dependencies among changes in the change history which are so important for backtracking. We have thus much to learn by "mining" change histories [7].

5. Acknowledgement

This research was funded by the Austrian FWF under agreement P21321-N15.

6. References

[1] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Communication of the ACM vol. 15, pp. 1053-1058, 1972.

[2] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development," International Journal on Software Engineering and Knowledge Engineering pp. 31-58, 1991.

[3] A. Egyed, "Instant Consistency Checking for the UML," presented at Proceedings of the International Conference on Software Engineering (ICSE) 2006.

[4] H. Gall, "Of Changes and their History: Some Ideas for Future IDEs," in Proceedings of the 2008 15th Working Conference on Reverse Engineering - Volume 00: IEEE Computer Society, 2008.

[5] R. Robbes and M. Lanza, "A Change-based Approach to Software Evolution," Electronic Notes in Theoretical Computer Science, vol. 166, pp. 93–109, 2007.

[6] A. Egyed, "Fixing Inconsistencies in UML Design Models," presented at Proceedings of the International Conference on Software Engineering 2007.

[7] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," presented at 28th International Conference on Software Engineering, Shanghai, China, 2006.