# Selective and Consistent Undoing of Model Changes

Iris Groher and Alexander Egyed

Johannes Kepler University Linz
Altenbergerstr. 69, 4040 Linz, Austria
{iris.groher,alexander.egyed}@jku.at

**Abstract.** There are many reasons why modeling tools support the undoing of model changes. However, the sequential undoing is no longer useful for interrelated, multi-diagrammatic modeling languages where model changes in one diagram may also affect other diagrams. This paper introduces selective undoing of model changes where the designer decides which model elements to undo and our approach automatically suggests related changes in other diagrams that should be undone also. Our approach identifies dependencies among model changes through standard consistency and well-formedness constraints. It then investigates whether an undo causes inconsistencies and uses the dependencies to explore which other model changes to undo to preserve consistency. Our approach is fully automated and correct with respect to the constraints provided. Our approach is also applicable to legacy models provided what the models were version controlled. We demonstrate our approach's scalability and correctness based on empirical evidence for a range of large, third party models. The undoing is as complete and correct as the constraints are complete and correct.

## 1 Introduction

We believe that the very nature of software modeling is about exploring design alternatives by trying out ideas and dismissing them if they are not satisfactory. However, today, modeling languages solely capture the final state of the model of a software system but fail to remember the many changes made along the way (the design history [9] with its decisions [13]). To compensate, modeling tools provide undo or version control mechanisms. However, these mechanisms capture the history of changes chronologically and if an undo is desired then a designer is forced to undoing changes chronologically (also undoing unrelated, intermittent changes).

This paper presents an approach for the selective undoing of design changes during software modeling where previously discarded changes can be recovered without having to undo unrelated, intermittent changes. Selective undoing is particularly important during multi-view modeling (such as the UML with its many diagrammatic views) because logically related model elements are intentionally spread across independent diagrams to separate concerns [14]. Recovering a discarded model element may then require the recovering of model elements in other diagrams – or else risk causing inconsistencies in the model.

D.C. Petriu, N. Rouquette, Ø. Haugen (Eds.): MODELS 2010, Part II, LNCS 6395, pp. 123–137, 2010.

Unfortunately, designers do not explicitly capture all dependencies among model elements – nor would it be feasible to expect them to do so. It would also be invalid to expect related model elements to be in "close proximity" (time and space). Related model changes could be done independently by multiple designers – at different times and in different diagrams. Likewise, single designers may concurrently perform multiple unrelated model changes. Any heuristic that was to infer relationships among model elements based on the time the changes happened or their location would be fundamentally flawed and useless.

This paper demonstrates that it is possible to automatically recover hidden dependencies among model changes through the help of consistency rules, well-formedness rules, and other design constraints [5] that typically exist and are enforced during software modeling. Constraints are specific to the modeling language or application domain. But our approach should be applicable to any modeling language for which such constraints are definable. In our experience, most modeling languages have such constraints and constraints are freely definable by the user.

We previously demonstrated how to detect [5] and fix [6] inconsistencies in design models – but their use for recovering hidden dependencies among model changes is new. The designer first selects previously discarded versions of model elements for undoing (manual input) and our approach automatically evaluates whether the undoing of the designer-selected versions causes inconsistencies. If it does then our approach recursively tries to resolve the inconsistencies by automatically considering other undoing choices that have the potential of fixing the inconsistencies. If it finds them then our approach informs the designer of what other model elements to undo to avoid inconsistencies.

Our approach is well-defined and precise. Its computational efficiency and scalability were evaluated through the empirical analysis of large, third-party, industrial software models. Since it detects dependencies among model elements based on model constraints, the quality of the undoing is as complete as the defined model constraints are complete. Correctness was assessed by validating whether the approach was able to recover a consistent state of the model after undoing (if it ever existed). Of course, our approach does not presume the models to be fully consistent [16]. Pre-existing inconsistencies are simply ignored.

## 2 Illustrative Example

To illustrate the benefits of the selective undoing of model changes, consider the video-on-demand system in Figure 1. The left and middle models represent two existing model versions (snapshots), each containing three different diagrams (structural, scenario, behavioral). The top diagram of version 1 shows the static structure of the system in form of a class diagram: the display responsible for visualizing videos and receiving user input; the streamer responsible for downloading and decoding of video streams. The middle diagram presents a particular usage scenario in form of a sequence diagram. This diagram describes the process of selecting a movie and playing it. Finally, the bottom diagram shows the behavior of the streamer in form of a statechart diagram (a toggling between two states).

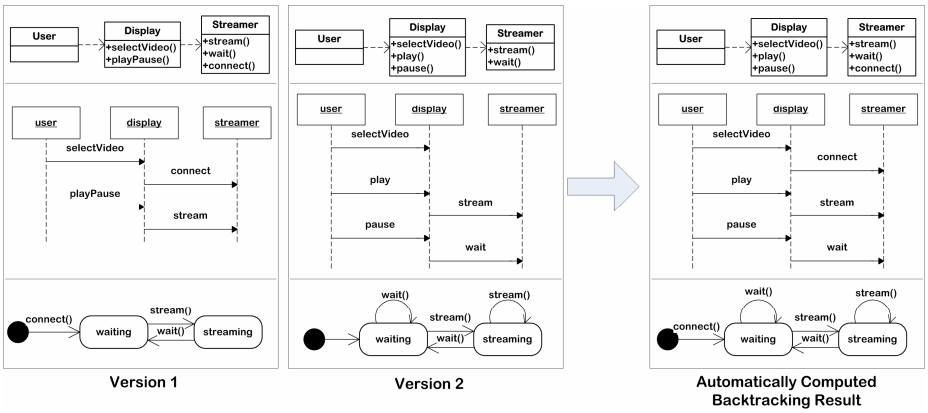


**Fig. 1.** Two Versions of the Video-on-Demand System and Result of Undoing connect

Version 2 (Figure 1 middle) depicts a later design snapshot of the video-on-demand system. The designer has made several design changes. It has been decided that having a single method for both playing and pausing movies is no longer desired. Instead, the designer renamed the *playPause()* method to *play()* and created an additional method named *pause()*. The sequence diagram was adapted accordingly. Additionally, the *connect()* method in *Streamer* was deleted and the behavioral diagram of the *Streamer* and the scenario were adapted (e.g., selecting a movie no longer requires explicitly connecting to the streamer).

The illustration does not depict logical dependencies among model elements in the different diagrams explicitly; however, they are present in form of model constraints. These constraints (often called consistency rules) describe conditions that a model must satisfy for it to be considered a valid model. Table 1 describes two such constraints on how UML sequence diagrams relate to class and statechart diagrams. Constraint 1 requires the name of a message to match an operation in the receiver's class. If this constraint is evaluated on message *stream* in the sequence diagram in Figure 1 (version 1) then it first computes all operations of the message's receiver class. The receiver of the *stream* message is the object *streamer* of type *Streamer* and the class' methods are *stream(), wait(),* and *connect().* The constraint is satisfied (i.e., consistent) because the set of operation names in *Streamer* contains one with the name *stream* – the name of the message. Constraint 2 states that the sequence of messages in the sequence diagram must correspond to allowed events in the statechart that describes the behavior of the receiver's class. The given UML models are internally consistent (for simplicity) which is not required by our approach.

**Table 1.** Sample Constraints (taken from literature)

| | |
|---|---|
| **Constriant 1** | **Name of message must be declared as operation in receiver class**<br>operations=message.receiver.base.methods<br>return(operations->name->contains(message.name)) |
| **Constraint 2** | **Sequence of messages must correspond to events**<br>start=state transitions that correspond to first message<br>return (start->exists(message sequence equal reachable sequence from start)) |

Now imagine that the designer desires to recover the state of the model where the *connect()* message in the sequence diagram still existed (from version 2 to version 1) – without having to undo other changes that were made since (e.g., separate operations *play* and *pause*). The right of Figure 1 presents the desired result of this undoing which is a compromise between version 1 and version 2. It illustrates the challenge of selective undoing because simply recreating the *connect* message is insufficient since it causes two inconsistencies (no such operation exists in *Streamer* and no such transition exists in the statechart because both were deleted in version 2). Because of the logical dependency between this message in the sequence diagram and the operations defined in the message's receiver class (cf. Constraint 1 in Table 1) we also need to undo the *Streamer* class to a version where the *connect()* operation existed. Further, because of the dependency between the *connect()* message and the transitions in the statechart (cf. Constraint 2 in Table 1) we need to undo the first transition to a version where it was named *connect*. However, we do not wish to undo the many other, unrelated changes. For example, the decision to change the *playPause()* operation in version 1 into separate operations *play()* and *pause()* in version 2 should not be affected by the undoing of the *connect()* message.

## 3 Related Work

Existing version control systems such as CVS [1] or Subversion [2] support undoing of entire models to any version but the granularity is typically to coarse grained to support selective undoing of individual model elements. Even a finer-grained version control system would not solve the problem because such systems are not able to automatically infer logical dependencies among model changes. Undo mechanisms as provided by most modeling tools are much more fine-grained than versioning systems. However, their change history is typically not persistent and the undoing is purely chronological, which causes the undo of intermittent changes which may not be related.

Selective undoing requires exploring the different versions of model elements that have existed in the past. To that extent, our approach treats the model elements with versions as "variables" and tries to set them such that inconsistencies are minimized. The constraint satisfaction problem (CSP) [10] and MaxSAT are thus related to our approach. In CSP, a constraint-based problem comprises a set of variables and a set of constraints across the variables. Solution techniques (CST) are capable of computing feasible values for all variables for all constraints (i.e., versions that satisfy all the consistency rules). Indeed, our approach makes use of how CST eliminates infeasible choices but not how the remaining ones are validated. Our approach also borrows from existing optimizations such as the AC3 optimization [11] which maps choices to affected constraints to efficiently determine what part of a model to re-evaluate when it changes (i.e., change=undoing). Unfortunately, CSP typically does not scale, especially not for large software models.

Truth maintenance systems (TMS) [4] focus on facts that make constraints hold. This is similar to our approach where logical dependencies among changes are captured to allow for an automatic recovery during undoing. TMS require the existence of the relations in advance which is not the case in our approach. The

dependencies are recovered automatically by observing constraints during their evaluation. Also TMS require the impact of constraints to be modeled from all perspectives (all kinds of changes) which is traditionally not done for modeling languages and thus it would be impractical to apply TMS here.

The approach by Mehra et al. [12] supports differencing and merging of visual diagrams. Semantic problems that arise during the merging process can be detected (which is similar to our approach) but not resolved automatically. Furthermore, our goal is not to fix inconsistencies during undoing that have existed in the past nor does it require doing so. Thus our work supports *living with inconsistencies* [3, 8].

# 4 What Is the Problem?

## 4.1 Definitions and Input

As input, our approach requires a model and a *change history*. Our approach either computes the change history by comparing different model versions (offline or legacy models) or by monitoring the designer and recording the model changes (online). For undoing, a designer must select a *Location* (or set thereof) which is a field of a model element (*ModelElementFields*) and must choose an earlier version. For example, fields of a UML class include its name, owned operations, and attributes. A *Version* is thus a value for a *Location* (cf. Figure 2).

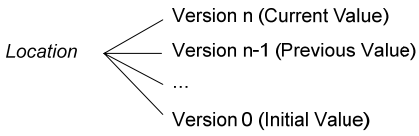


**Fig. 2.** A Location, its Current Value, and previous Values

The list of all available locations for undoing is thus the list of all fields of model elements in the model where the number of versions is greater than 1. An undo is simply the assignment of a *Version* to a *Location* (only one version can be assigned to a location at any given time) although different locations could be assigned different versions (the following uses a syntax similar to OCL).

$$Location \in ModelElementFields$$

$$Locations := ModelElementFields \rightarrow select(l : Location \mid l.versions \rightarrow size() > 1)$$

## 4.2 User Actions and Impact

The user initiates the undoing by selecting one or more locations and versions. For example, in the video-on-demand illustration in Section 2, the user chooses to restore the *connect()* message. The approach automatically creates, modifies, or deletes the model element(s) selected for undoing. If the model element was deleted then undoing must re-create it and modify it to reflect the desired version (i.e., all its fields must be set in addition to creating the model element). If the element still exists then

it must be modified or deleted. For undoing the *connect* message, we need to re-create the message, name it *connect*, and restore its other fields (e.g., including the ones that map it to a specific location in the sequence diagram). The undoing changes the model, which, in turn, affect the consistency of the model. User-induced undoing thus may cause inconsistencies because:

1. *Incomplete undos*: Caused by an incomplete selection of locations and versions. Additional locations need to be undone to solve the inconsistencies and the user needs to be given choices on what else to undo also.
2. *Incompatible undos*: The user selected locations and versions are incompatible. In this case the inconsistencies cannot be resolved and the user needs to be notified of the incompatibility.

Existing technologies for consistency checking [5] are able to identify inconsistencies caused during undoing. But consistency checking alone is not the solution to the problem. We also need to identify which combinations of other undoing resolves the inconsistencies.

### 4.3 Naïve but Unscalable Solution

To solve the problem we could automatically try other undos to see whether they resolve the inconsistencies caused by the user-initiated undoing. The tricky part is to find some combination of additional undos that resolve the inconsistencies caused by the user actions without causing new inconsistencies.

A correct and simple solution would be to try all possible combinations of locations and their versions available in the change history (brute force solution). The computational complexity of such a brute force evaluation is unfortunately $O(\#C * \#versions^{\#locations})$ – exponential with the number of locations as the exponential factor and the number of versions as the exponential base. *#C* represents the total number of constraints imposed on the model. Table 2 illustrates the futility of the brute force approach on five, large UML models. *# Snapshots* shows the number of model snapshots we compared. *# Changes* shows the total number of model changes from version 1 to version n (note only those changes are included that are processed by the constraints included in this study). *# Locations* shows the number of locations the model consists of. *# Combinations* shows the number of combinations we would need to evaluate when using a naïve, brute force approach for selective undoing ($AV^{\#Changes}$ where $AV$ is the average number of versions per location that has changed).

**Table 2.** Computational Complexity

| | **# Snapshots** | **# Changes** | **# Locations** | **# Combinations** |
|---|---|---|---|---|
| caBIO | 3 | 101 | 11,422 | $2.81^{101}$ |
| Calendar | 2 | 73 | 17,943 | $2.0^{73}$ |
| UMS | 5 | 98 | 14,874 | $3.15^{98}$ |
| Flipper | 4 | 55 | 6,338 | $3.69^{55}$ |
| anonymous | 2 | 104 | 156,572 | $2.0^{104}$ |

The table shows that the number of combinations (*# Combinations*) is unmanageable regardless of model size. The next section presents our solution.

## 5 Undoing Model Changes

In order to efficiently undo design changes we have to be able to:

1. detect inconsistencies caused during undoing quickly and
2. identify other model elements whose undoing resolve these inconsistencies

Fortunately, part 1 was already solved by our previous work [5]. We use the Model/Analyzer approach for instant consistency checking to detect inconsistencies caused by the user-triggered undo. Part 2 (the automatic fixing of inconsistencies caused by exploring previous model element versions) is new and thus the focus of the remainder of this paper. As part 2 builds on top of part 1, we briefly summarize our solution to part 1 in the next section.

### 5.1 Part 1: Incremental Consistency Checking

The Model/Analyzer approach [5] (previously known as UML/Analyzer) is capable of quickly and correctly evaluating the consistency of models after changes. This approach treats every evaluation of a constraint separately. It essentially instantiates a constraint as many times as there are model elements in the model that must be evaluated by that constraint. For example, constraint 1 in Table 1 must be evaluated 4 times for version 1 in Figure 1 – once for every message. The Model/Analyzer approach thus maintains 4 constraint instances (*C1_selectVideo*, *C1_playPause*, *C1_connect*, *C1_stream*). All 4 constraint instances are evaluated separately as they may differ in their findings (although all are currently consistent). To support incremental consistency checking, the approach monitors the behavior of the consistency checker to identify which model elements a constraint instance accesses during its evaluation. If one or more of these model elements change then the constraint instance must be re-evaluated. For example, constraint 1 in Table 1 compares message names and operation names. However, it does not randomly access operations and messages. Instead, the constraint starts at a predefined message and navigates the model to first identify the message receiver object (UML model element of type Lifeline), next accesses the class that this object instantiates, and finally accesses the operations of that class. The scope of the constraint is that a subset of *ModelElementFields* and consists of the message itself, its receiver, the receiver's base class, and the operations of this class. On a concrete example (cf. Figure 1, version 1), the evaluation of the constraint instance *C1_playPause* accesses the message *playPause()* first, then navigates to the message's receiver object *display*, its base class *Display*, and finally the methods *selectVideo()*, and *playPause()*. This list of model elements accessed during the evaluation of constraint instance *C1_playPause* is defined to be the scope of this constraint instance – it is observable automatically. Obviously, if the message name changes, the operation name changes, or certain other model elements part of the scope change, then the consistency is in jeopardy. In such a case, constraint *C1_playPause* is affected by the change and must

be re-evaluated. The evaluation of other constraint instances have different (but often overlapping) scopes. Using the Model/Analyzer approach we can compute the constraint instances affected by a location. It is simply all constraint instances where the location is part of their scope.

$$AffectedConstraints(l : Location) := Constraints \rightarrow collect(c : Constraint \mid c.scope \rightarrow includes(l))$$

The changes caused during undoing thus trigger automated re-evaluations of all those constraints that contain the changed location(s). With the creation and deletion of model elements, constraints must also be re-instantiated or disposed of. This capability also existed and was discussed in [5, 6].

The undoing is complete if the changes triggered do not cause inconsistencies. On the other hand, if the undoing of model elements (i.e., the deletion, creation, or modification) causes inconsistencies then further undoing may be necessary. In the example, the undo of the message *connect()* to a version where it existed causes two inconsistencies because the *Streamer* class no longer contains the corresponding operation with the name *connect()*. Also, the statechart no longer contains a transition with the name *connect()*. The undoing thus violates two constraint instances of constraints 1 and 2 defined in Table 1 (it is important to observe here that with constraint instances we are in essence referring to the model elements in the scope and no longer to the types of model elements). Table 3 reveals that a constraint is problematic if it was consistent before the undoing but no longer is thereafter; or if the undoing causes the instantiation of a constraint that is then inconsistent. Both cases imply that the undoing may have been incomplete (i.e., other model elements may also need to be undone) or that the undoing may be composed of incompatible locations. This distinction is explored later.

**Table 3.** Effects of Undoing Changes on Constraints

| **Constraints** | | **After** | | |
|---|---|---|---|---|
| | | Consistent | Inconsistent | Disposed |
| **Before** | consistent | no problem | **problem** | no problem |
| | inconsistent | no problem | no problem | no problem |
| | disposed | no problem | **problem** | no problem |

### 5.2 Part 2: Incremental Version Exploration

If the undoing causes inconsistencies then our approach investigates these inconsistencies and attempts to fix them by exploring additional locations to undo that would resolve them (first individually, then combined). We initially presume that an inconsistency caused during undoing is caused due to incomplete undoing. Our approach thus searches for additional locations to undo such that they resolve the inconsistency at hand. If no such locations can be found then the inconsistency must have existed previously and thus cannot be resolved; or the user-selected undo included incompatible versions (only applies if the user selected two or more locations for undoing).

Figure 3 presents a sketch of the algorithm. The approach first changes the user-selected locations to the versions selected, The Model/Analyzer approach will identify

all constraint instances (in the following denoted as constraints for brevity) affected by this change and also instantiate new constraints if so needed. The *affectedConstraints* collection includes all these constraints. The algorithm then iterates over all affected constraints, evaluates them one by one, and, if the consistency was affected, adds them to the *inconsistencies* collection. If inconsistencies is empty then the undoing is complete. For all inconsistent constraints, additional undoing is necessary to fix them. In our example of undoing the *connect* message, the Model/Analyzer approach identifies the constraints *C2_streamer* and *C1_connect* as affected. *C2_streamer* refers to an existing constraint whereas *C1_connect* was instantiated because the *connect* message was re-created. Both constraints are inconsistent after undoing and both constraints needed to be investigated further to identify additional locations for undoing.

```
undoSelectedVersions (selectedVersions)
     for all selectedVersions
          affectedConstraints = change(selectedVersions)
          for all constraint:affectedConstraints
               if (not validate(constraint)) inconsistencies.add(constraint)
     end
     if (inconsistencies.size>0) undoAdditionalVersions(inconsistencies)
undoAdditionalVersions (inconsistentConstraints)
     for all constraint:inconsistentConstraints
         locations = validate(constraint)
               for all additionalVersions: locations x versions
                    change (additionalVersions)
                     if (validate(constrain))
                         validAssignment=additionalVersions
               end
         end
     end
     affectedConstraints = IntersectionValidAssignments (validAssignments)
     if (affectedConstraints.size>0)
                         undoAdditionalVersions(affectedConstraints)
```

**Fig. 3.** Undoing Selected Versions

In [6] we showed how to narrow down the search for fixing inconsistencies. In essence, an inconsistency can be fixed only by changing one or more model elements that the inconsistent constraint accessed during its evaluation. We already identified this list of accessed elements as the *scope* of a constraint (recall Section 2). To fix inconsistency C1, we would have to change one or more of these scope elements. However, only those scope elements can be changed for which we have alternative versions available.

$$Locations(c:Constraint) := Locations \cap c.scope$$

The locations for fixing an inconsistency caused during undoing are simply the intersection of all valid locations (=model elements for which multiple versions are available) and the scope elements for that constraint. The *undoAdditionalVersions* algorithm explores this. The algorithm iterates over all inconsistent constraints (in the first run those are the ones identified in the *undoSelectedVersions* function in Figure 3) and identifies which locations the constraint accesses during its evaluation The algorithm then explores the cross product of all possible fixes for each

inconsistent constraint. That is, the algorithm tries all combinations of model versions for this subset of locations only. An *Assignment* for a *Constraint* is one such exploration where we have to set a *Version* for every *Location* encountered by that constraint (excluding the user-selected versions which are fixed).

$$Assignment(c : Constraint) := Locations(c) \rightarrow collect(l : Location \mid random(l.versions \rightarrow value))$$

A constraint exploration is then simply the function of an assignment onto a truth-value such that the truth value reflects the consistency of the constraint for the given assignment. A valid assignment is defined as an assignment where the constraint evaluates to true.

$$ValidAssignment(c : Constraint, a \in Assignment(c)) := Evaluation(c, a) = true$$

The *ValidAssignments* (if size>0) represents additional undos (=model changes) to fix the inconsistencies caused by the initially user-selected undoing. A ripple effect. refers to the situation where the additional undos may, in turn, affect additional constraints. In other words, the inconsistencies caused by the user-selected undoing can only be resolved by undoing additional model element, which in turn, may cause more inconsistencies with respect to their affected constraints. This ripple effect is recursive and terminates only once no more inconsistencies are caused. All valid assignments that contain versions that differ from the last version require additional undos. The set of affected constraints thus needs to be incrementally expanded. The last lines of the *undoAdditionalVersions* function selects the assignments consistent with all affected constraints, determines which constraints are affected by these assignments, and then computes the ripple effect via recursive descend. The recursive descend terminates when no more inconsistencies are encountered.

Figure 4 illustrates how versions and constraints are incrementally explored for the example introduced in Section 2. The user chose to undo the *connect* message. Two constraints are initially affected (the instantiated constraint *C1_connect* and the existing constraint *C2_streamer*) because both evaluate to false and thus need further undoing. During evaluation of *C1_connect* and *C2_streamer* new locations are incrementally instantiated (with all their versions as choices). For *C1_connect*, there is one such location only: *base[operations]*. Note that in Figure 4, only the locations that have versions are displayed. Both versions available for *base[operations]* are thus explored and constraint *C1_connect* is evaluated separately for each one. The assignment {*stream*, *wait*, *connect}* is valid because it contains the missing operation *connect.* The assignment {*stream*, *wait*} is not valid as it misses that method. The list of valid assignments for constraint *C1_connect* thus contains *{stream, wait, connect}*.

Because the chosen valid assignment contains an older version of the location *base[operations],* we now need to investigate which constraints are affected by the change of this location (this is no longer a user-selected location and thus the initially computed list of affected constraints is no longer complete). Two additional constraints are affected: both *C1_wait* and *C1_stream* contain this *base[operations]* in their respective scopes and both need to be evaluated now also. Both were consistent before the undoing of *base[operations]* and after re-evaluating them, we find that both a still consistent. This means that the undoing of *base[operations]* did not negatively impact these additional constraints and no further undoing is required. The recursion stops here. If they would have been inconsistent then we would have had to try the versions of all locations in scope of *C1_wait* and *C1_stream*.

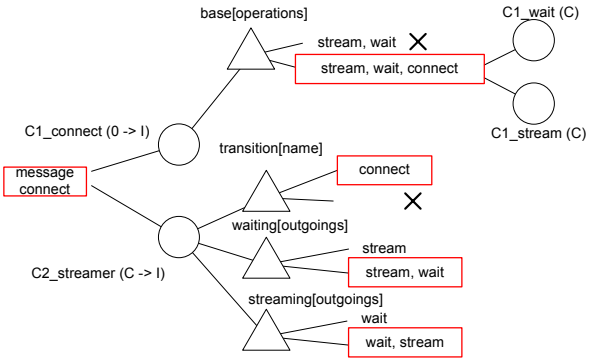


**Fig. 4.** Incremental Version and Constraint Exploration

Thus far, we only dealt with one of the two initially affected constraints. The other initially affected constraint was *C2_streamer* which must be explored next. In its scope are not one but three locations for undoing. Each location in Figure 4 is indicated as a triangle and each location has two versions. There are thus $2^3$=8 possible assignments and constraint *C2_streamer* is explored for all of them. Of these eight assignments, four are valid assignments ({*connect*, *stream*, *wait*}, {*connect*, *stream*, *wait-stream*}, {*connect*, *stream-wait*, *wait*}, {*connect*, *stream-wait*, *wait-stream*}). So, we have several options here. The value of the location *transition [name]* needs to be set to *connect*. The locations *waiting[outgoings]* and *streaming[outgoings]* both have 2 correct choices. Since our goal is minimal undoing, we prefer to choose the latest versions if we have multiple valid choices. The latest versions of the two locations (the assignment with the cumulative least amount of undoing) are selected ({*stream*, *wait*} and {*wait*, *stream*}). A detailed description of how assignments are selected is given in the next section. No further constraint is affected by those locations. The resulting model after all undos corresponds to the right of Figure 1.

## 6 Selective Undo with the IBM Rational Software Modeler Tool

Our approach is fully tool supported and integrated with IBM Rational Software Modeler (RSM) design tool and our Model/Analyzer consistency checking tool. The consistency checker receives change notifications from RSM and implements the instant consistency checking and fixing of inconsistencies as described in [5-7]. The consistency rules are written in Java and OCL. The tool also supports the computation of change histories for legacy models as wells as the recording of fine-grained change histories.

## 7 Validation

This section demonstrates that our approach scales - even for models with tens of thousands of elements. We empirically validated our approach on five versioned,

third-party UML models and 22 types of consistency and well-formedness rules taken from literature. Table 2 already listed the models which differ substantially in model size and types of model elements used.

### 7.1 Computational Complexity

The total number of possible assignments for a constraint is the cross-product of all versions for all locations encountered by that constraint. The computational complexity of such an exploration is O(AC * #versions$^{\#locations}$) – or exponential with the number of locations as the exponential factor and the number of versions as the exponential base. *AC* represents the number of constraints affected by an undo. While exponential growth is daunting in general, we will demonstrate next through extensive empirical evidence, that in context of single constraint instances, both #versions and #locations are very small *and do not increase with the model size*.

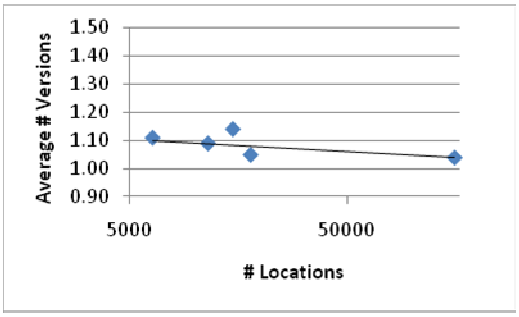


**Fig. 5.** Average Number of Versions per Location

### 7.2 Scalability Drivers

We measured the number of locations (*#locations*) of over 140.000 constraint instances [5] across all five models. There exist a wide range of values between the minimum and maximum number of locations *but the averages stayed constant with the size of the model*. Over 95% of all 140.000 constraint instances evaluated less or equal than 25 model elements which is an important scalability factor because it implies that the exponential factor is a constant.

It is also important how many constraints are affected by an undo (*AC*). In [5] we computed the number of constraints affected by a single change. We again found a wide range of values between the smallest and largest number of constraints but *the average also stayed constant with the model size*. Our evaluations showed that in average only 1-10 constraints had to be evaluated.

Now that we have seen that both the number of locations (#locations) and the number of affected constraints (*AC*) are small values that stay constant with the size of the model, we need to look at the remaining scalability factor *#versions*: the number of versions per location. Figure 5 depicts the average number of versions per location for all five models which also appears to stay constant with the size of the

model with 1.09 versions per location. Indeed, the likelihood for a location to have versions decreases exponentially with the number of versions (data omitted for brevity). So, while the evaluation of a constraint is exponentially complex within its locations and versions, the fact that all exponential factors are small and do not increase with the size of the model implies that our approach scales.

There is however another potential scalability problem: the incremental exploration of affected constraints. Our approach only investigates constraints if they directly relate to changes caused by undos. If an undo causes inconsistencies then further changes are necessary which may uncover additional, affected constraints. The exploration, which we referred to as the ripple effect, may in theory snowball into a very large number of incrementally affected constraints where, perhaps, the exploration of individual constraints is scalable but not the ripple effect. In Figure 6, we thus empirically evaluated the average impact of the ripple effect on all five models. We found that the initial number of affected constraints was between 3 and 4.5 constraints, depending on model but that this number decreased in average with every ripple and always terminated before the 4 ripple (note: a ripple is a recursive descend where a change affects constraints which need to be changed which affects more constraints which…)

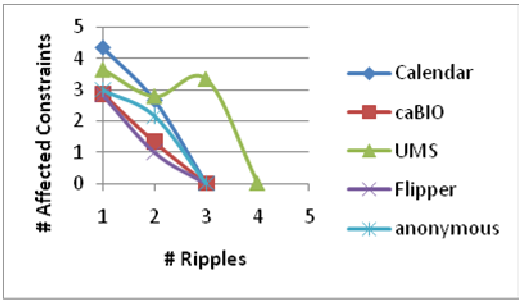


**Fig. 6.** Ripple Effect

### 7.3 Correctness

Whenever inconsistencies were caused by the undoing of model changes, we also explored (in both a brute force manner as well as our approach) whether they were resolvable and could be resolved by our approach. We found in all cases that our approach was able to compute a consistent undo if such a consistency had existed in the past. Interestingly, our approach sometimes also resolved inconsistencies that had always existed. It is unclear to us whether this is a benefit or whether intentionally unresolved inconsistencies should remain so (i.e., living with inconsistencies). This feature could be disabled if needed.

### 7.4 Memory Consumption

In [5] we found that there exists a linear relationship between the model size and the memory cost for storing the scopes. The memory cost rises linearly with the number

of constraint instances and is *O(#constraints * scope size)*. The storage of the change history is also manageable because only few model elements change. Since we have 1.09 versions per location, it follows that the memory cost for the change history is 1.09 times the model size plus overhead.

### 7.5 Threats to Validity

As any empirical study, our exploratory experiments exhibit a number of threats to validity [15]. A threat to *construct validity* – are we measuring what we mean to measure? – is the potential that our validation may underrepresent the construct. We validated our approach on 5 large-scale, industrial models with tens of thousands of model elements and up to 5 versions. The models and their versions cover years of development and we thus believe that they represent typical undoing scenarios found in industry. The threat to *internal validity* – are the results due solely to our manipulations – is selection, in particular the selection of the models and the consistency rules. The models are different in size and domain and our approach performed well in all models. Also, the 22 selected consistency rules are covering nearly a complete set of dependencies between UML class diagrams, sequence diagrams, and state charts. Regarding *conclusion validity* we have seen that our approach scales for large models up to 150.000 locations. With respect to *external validity* – can we generalize the results – we took real-world large models representing realistic application contexts. Our empirical validation does not definitively proof that more versions are possible per model elements but it does confirm that most model elements never change (these models and their versions cover years of development). Even if the actual number of versions is higher, it is not a problem because, in general, few model elements change. However, one could argue that when using versioned models, we do not always see all changes because a location may change multiple times between versioned model snapshots. Yet, the versions we used represent major milestones of the system under development and missing intermediate versions, if they did exist, are likely less interesting results or else they likely would have been version controlled. The biggest threat to external validity, however, is that we did not yet assess the usability of our approach by monitoring and interviewing engineers that used our tool. This is part of our future work. We plan to evaluate how difficult it is for users to manually fix inconsistencies introduced during selective backtracking compared to using our tool.

## 8 Conclusion

This paper discussed an approach for the selective undoing of design changes. Designers can explore earlier alternatives concurrently and undo them independently if needed. This is also beneficial if multiple designers are working on the same model and want to undo changes without necessarily undoing other designer's changes. Selective undoing of changes is a difficult problem because of the complex, logical dependencies among design changes. We solved this problem by automatically discovering dependencies among versions of model elements through the help of consistency rules. We demonstrated on five case studies that our approach scales and

produces correct results. Our approach does not require a consistent model as input. Neither is it limited to certain constraints only.

**Acknowledgments.** We like to thank Alexander Reder for his help on the tool. This research was funded by the Austrian FWF under agreement P21321-N15.

## References

[1] Concurrent Versions System (2009), http://www.nongnu.org/cvs/
[2] Subversion (2009), http://subversion.tigris.org/
[3] Balzer, R.: Tolerating Inconsistency. In: Proceedings of 13th International Conference on Software Engineering (ICSE), pp. 158–165 (1991)
[4] Doyle, J.: A Truth Maintenance System. Artificial Intelligence 12, 231–272 (1979)
[5] Egyed, A.: Instant Consistency Checking for the UML. In: Proc. of the 28th Intern. Conf. on Software Engineering, Shanghai, China, pp. 381–390 (2006)
[6] Egyed, A.: Fixing Inconsistencies in UML Design Models. In: Proceedings of the 29th International Conference on Software Eng., pp. 292–301 (2007)
[7] Egyed, A., Wile, D.S.: Support for Managing Design-Time Decisions. IEEE Transactions on Software Engineering 32, 299–314 (2006)
[8] Fickas, S., Feather, M., Kramer, J.: Proceedings of ICSE 1997 Workshop on Living with Inconsistency, Boston, USA (1997)
[9] Gall, H.: Of Changes and their History: Some Ideas for Future IDEs. In: Proc. of 15th Working Conf. on Reverse Eng., Antwerp, Belgium, p. 3 (2008)
[10] Henteryck, P.: Strategic Directions in Constraint Programming. ACM Computing Surveys 28 (1996)
[11] Mackworth, A.K.: Consistency in Networks of Relations. Journal of Artificial Intelligence 8, 99–118 (1977)
[12] Mehra, A., Grundy, J., Hosking, J.: A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In: Proceedings of the 20th International Conference on Automated Software Engineering (ASE), Long Beach, CA, pp. 204–213 (2005)
[13] Robbes, R., Lanza, M.: A Change-based Approach to Software Evolution. Electron. Notes Theor. Comput. Sci. 166, 93–109 (2007)
[14] Tarr, P., Osher, H., Harrison, W., Sutton Jr., S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proceedings of the 21st International Conference on Software Eng., pp. 107–119 (1999)
[15] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Dordrecht (2000)