Inconsistency Resolution in Merging Versions of Architectural Models

Hoa Khanh Dam
University of Wollongong
New South Wales, Australia
hoa@uow.edu.au

Alexander Reder
Johannes Kepler University
Linz, Austria
alexander.reder@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

***Abstract*—State-of-the-art optimistic model versioning systems, which are critical to enable efficient team-based development of architectural models, are able to detect and help resolve basic conflicts arising during the merging of model versions. However, it is often overlooked that model merging may also cause severe syntactical and semantic inconsistencies. In this paper, we propose an approach to guide the resolution of inconsistencies detected in a merged architectural model. Our approach automatically finds and presents to the software architects all solutions for resolving all inconsistencies arisen during the merging of model versions. For inconsistencies that pre-exist in the model, our approach is able to suggest exactly which model elements should be changed to resolve them. Our approach is built upon a repair generation which can quickly derive resolutions for an inconsistency by examining its static and dynamic structure and forming concrete repair actions from changes in the versions to be merged. An empirical validation on a range of industrial models has demonstrated that our approach is scalable to both large models and large differences between model versions.**

I. Introduction

Architectural models have become central artifacts which are created and used by software architects. In a collaborative environment, which is the dominant form of today's software development, software architects concurrently and independently work on architectural models which subsequently need to be merged. A basic scenario is where multiple architects work independently on a single architectural model and, since they do so separately on their respective workstations, different versions of that model may exist. These different versions then need to be merged periodically to support collaboration and error detection among these architects. In another scenario, multiple versions of a model may exist due to the concurrent evolution of product variants. For example, a company may develop multiple related software products, each undergoing constant evolution, to meet their respective, ever-changing user requirements and environmental changes. Here, merging may be desired to consolidate different variants or simply to facilitate reuse among the variants. There are many more such scenarios where software architects find themselves confronted with concurrently evolving versions of architectural models [1]. All these scenarios pose the challenging need to merge these different versions of architectural models.

However, since models are complex, rich data structures of interconnected elements, traditional text-based versioning techniques and tools such as Git, Subversion, and CVS have *not* been successfully applied to model versioning [2]. Without adequate tool support, model merging may result in a syntactically and/or semantically inconsistent merged version. Therefore, inconsistency management is of vital importance in model merging. However, state-of-the-art model merging techniques have only focused on detecting inconsistencies in merging versions of models (e.g. [3, 4]) and there has been very little work in resolving such inconsistencies having arisen during model merging.

Resolving inconsistencies however is much more difficult than detecting them since the number of alternative repairs increases exponentially with the complexity of the consistency rule and the number of model element accessed [5]. While previous work has shown that abstract repairs (which merely identify the model elements that require repairing) are reasonably localized and scalable to compute, concrete repairs (which identify all possible ways of repairing a given model element) are often infinitely large. For example, even if a repair merely requires the change of a single state transition action, we must consider that there are infinitely many ways of writing such actions. And, unfortunately, effective model merging needs to explore this apparently infinite space of concrete repairs for any inconsistency caused - an apparently computationally unfeasible endeavor.

In this paper, we argue that the space of concrete repairs for resolving inconsistencies is constrained by changes made in the versions to be merged. Indeed, we argue that in such a constrained space, model merging becomes practically feasible - not only in considering concrete repairs (as opposed to abstract repairs) but also in fixing a number of inconsistencies at once (as opposed to individual inconsistencies). However, we believe that consistent model merging may not be fully automated since there are decisions that involve tradeoffs where human expertise and communication are required. As a result, we propose an approach to assist the software architects in the merging process by suggesting alternative options in selecting which changes should be merged.

We extend previous work [5] in such a way that concrete repairs (i.e. a concrete value is known) for an inconsistency are automatically generated and actual side effects are computed (as opposed to abstract repairs as in [5]). Our approach generates complete and correct repair options for an inconsistency since it analyzes the structure of an inconsistency (i.e. the associated consistency constraint) as well as its expected and observed validation results to pinpoint exactly the cause of the inconsistency. Thus, our approach avoids the unnecessary computation cost of trying all the changes to see which combinations would fix an inconsistency as well as the correctness and completeness issues associated with hand-crafting resolution rules. Since considering a number of inconsistencies at once (those caused by the merging), our approach involves looking ahead to account for the side-effects of a repair (for an inconsistency) on the other inconsistencies that may exist. Our approach also identifies inconsistencies that are not caused by the merging and provides the software architects with guidance to resolve them. A number of empirical validations have shown that our approach is scalable to large industrial architectural models and large number of changes in the model versions to be merged.

II. Illustrative Example

We describe here a typical example of classical model merging where two software architects, Alice and Bob, concurrently work on developing an architectural model for a software controlling the washing machine. In this example, Alice and Bob use the Unified Modeling Language (UML) which has extensively been used for representing the architectural models of software systems in recent years [6, 7]. We however note that our approach also applies to arbitrary architectural models as long as they follow a well-defined metamodel with explicit constraints, which is today's norm.

Figure 1 shows a UML fragment of the architectural model which covers both the structural view (a class diagram) and the behavioral views (a sequence diagram and a state diagram). The class diagram describes three components *GUI*, *Control* and *Driver* and their connectors: an association (between *GUI* and *Control*) and a generalization (between *Control* and *Driver*). The sequence diagram describe a typical scenario of running the washing machine which involves the interaction between the instances of components *GUI* and *Control*, whereas the state machine diagram shows the behavior of the controller of the washing machine, i.e. component *Control*.

Let us now assume that both Alice and Bob check out the latest version (i.e. the Original Version) from a common repository and begin making their changes. Alice designs the new rinsing feature by adding a behavioral feature (in the form of an operation) *rinse()* in component *Control*, adding message *rinse* to component instance *ctrl* and adding state *rinsing* in the state machine diagram (see Version 1 in Figure 1). She also renames the message *start* to *run* and makes component *Driver* become a subtype of component *Control*. In the meanwhile, being unaware of Alice's changes, Bob completes the design for the stopping feature by making component *Control* become a subtype of component *Driver* and adding a message *stop* to component instance *ctrl* (see Version 2 in Figure 1). He also renames operation *run()* to *init()* in component *GUI*, renames message *start* to *init* (sent to component instance *gui*), and adds a message *turnOff* (sent to component instance *gui*).

When both Alice and Bob commit their own version, existing model versioning systems would typically produce a merged version as in Figure 1 and highlight a conflicting change: both Alice and Bob renamed message *start* differently, and would ask (either of) them to deal with the conflict. There are however several issues in this merged version in terms of the required syntactical (e.g. well-formedness) and semantical consistency (e.g. coherence between different views) for an architectural model. Such consistency conditions are usually specified in terms of constraints. Table I describes three typical consistency constraints on how a UML sequence diagram relate to class and state machine diagrams and the inheritance relationship between components in the class diagram. These three constraints are taken from the literature (C1 and C2 from [8]) and UML specifications (C3).

TABLE I
Example of consistency constraints

C1	The name of a message must match an operation in the receiver's component (the operation may be inherited from a generalization)
C2	The sequence of incoming messages to a component instance in a sequence diagram must match the allowed events in the state machine diagram describing the behavior of the component type.
C3	Inheritance cannot include cycles[1].

Since both Alice and Bob have each created an inheritance relationship between components *Control* and *Driver* but in different directions, both of them are integrated into the merged version which now has an illegal circulate inheritance (violating constraint C3). In addition, constraint C1*(start)* is violated in the original version (since message *start* received by instance *gui* of component *GUI* does not match with any operation in the component), and both Alice and Bob, each in their own way, have attempted to resolve this inconsistency. However, this constraint becomes violated again in the merged version since only the operation *run()* is updated and the conflict involving the renaming of message *start* is awaiting for manual resolution. In addition, in the merged version the new message *turnOff* does not match with any operation in component *GUI*, which causes another inconsistency (i.e. violation of constraint C1*(turnOff)*). Finally, there is still no operation in component *Control* matching with message *wash*, and thus inconsistency C1(*wash*) still exists in the merged version.

III. Understanding an Inconsistency

It is important to understand how inconsistencies have arisen in the merged model, as part of the investigation of how to resolve them. Table II captures a typical lifecyle of an inconsistency in terms of its *presence* (denoted as P) and *absence* (denoted as A) in a given version of the model. The

[1]Consistency constraints for UML are typically expressed in the standard Object Constraint Language (OCL). For instance, constraint C3 is expressed in OCL as *not self.allParents()→includes(self)* where *self* is the **context element**, i.e. the UML Class.

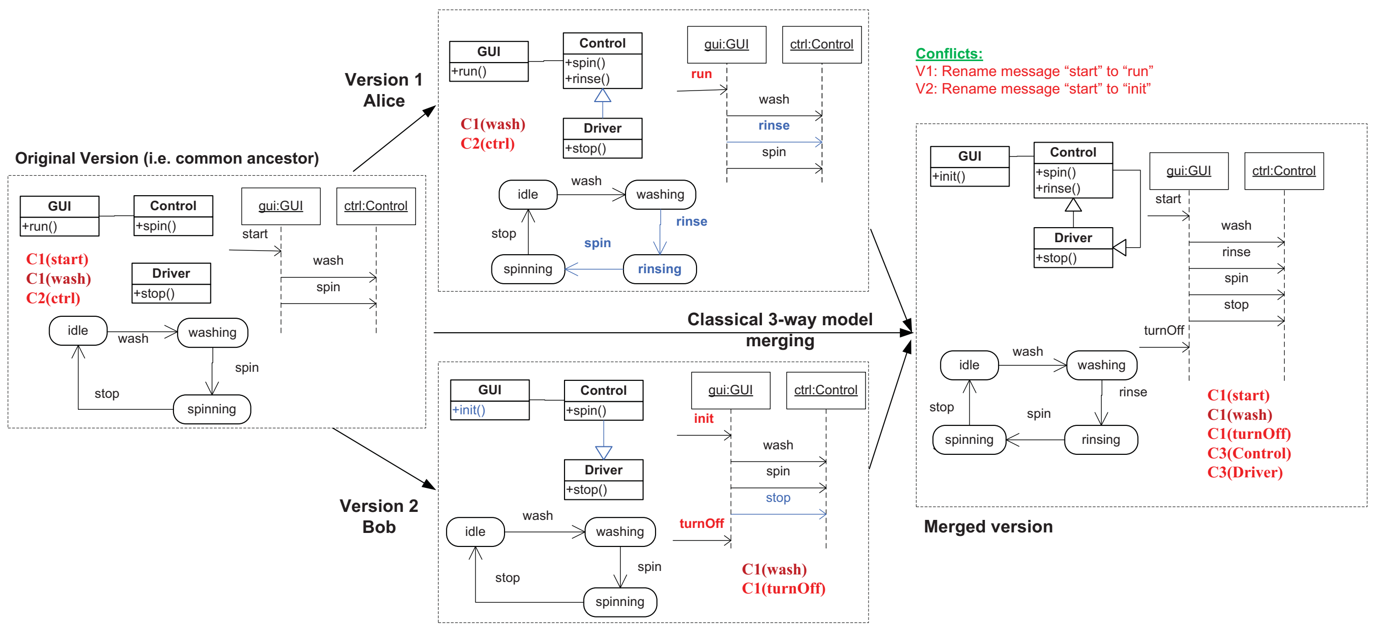


Fig. 1. An example of the traditional inconsistent model merging

TABLE II
LIFECYCLE OF AN INCONSISTENCY: BEING PRESENT (P) OR ABSENT (A)

Pattern	1	2			3			4	Other							
Original model	P	P	P	P	A	A	A	A	P	P	P	P	A	A	A	A
Version 1	P	P	A	A	P	P	A	A	P	P	A	A	P	P	A	A
Version 2	P	A	P	A	P	A	P	A	P	A	P	A	P	A	P	A
Initial merged model	P	P	P	P	P	P	P	P	A	A	A	A	A	A	A	A

presence of an inconsistency means that the corresponding consistency constraint has been instantiated and evaluated as being inconsistent. By contrast, the absence of an inconsistency indicates that the related constraint instance either has been evaluated as being consistent or has been destroyed (due to the deletion of the context element).

Inconsistencies existing in the original model may disappear in the merged model (e.g. $\mathsf{C2}(\mathit{ctrl})$ in the running example, see Figure 1) since the revised changes and/or the merging itself have fixed them (see column "Other" in Table II). The merged model may however contain inconsistencies due to one of the following patterns of reasons:

1) An inconsistency exists in the original model, still exists in the two versions (since neither of the changes were able to resolve it), and also exists in the merged model (since integrating the changes from both versions still cannot resolve it). The violation of constraint $\mathsf{C1}(\mathit{wash})$ is an example of this inconsistency type.
2) An inconsistency exists in the original model (e.g. $\mathsf{C1}(\mathit{start})$), but is absent in one or both revised versions (since either of the changes has fixed it), however returns in the merged model (since merging the changes has re-created the inconsistency).
3) An inconsistency (e.g. $\mathsf{C1}(\mathit{turnOff})$) does not exist in the original version, but is present in one or both revised versions (since the change(s) has caused it) and is still present in the merged model (since merging the changes has not affected it).
4) An inconsistency does not exist in the original model, still does not exist in both versions, but is present in the merged model (since merging the changes has caused the inconsistency). The violation of constraints $\mathsf{C3}(\mathit{Control})$ and $\mathsf{C3}(\mathit{Driver})$ is an example of this inconsistency type.

Inconsistencies whose lifecycle follow the first pattern have pre-existed in the original model and also existed in the versions. We will not be able to resolve them by reversing the revisions' changes since such changes were not the causes of the inconsistencies nor were able to resolve them. Applying conflicting changes, if they exist, may be able to resolve such inconsistencies. Otherwise, they are classified as *persistent inconsistencies*.

Inconsistencies whose lifecycle follow the remaining three patterns are caused by either the changes in the revisions (pattern 3) or the merging of those changes together (patterns 2 and 4). We will therefore be able to fix them by reversing the *appropriate* changes. Applying conflicting changes, if they exist, may also be able to resolve such inconsistencies, or create new inconsistencies, or has no effect on the consistency of the model. We refer to those inconsistencies as *non-persistent inconsistencies*.

Our approach detects inconsistencies using an existing incremental inconsistency checker [8] which identifies model elements that are changed and affect the truth values of consistency constraint instances. Such locations form the *scope* of a constraint instance, which is established by automatically observing which model elements are accessed during the evaluation of consistency constraints. For instance, the evaluation of constraint C1 on message *start* accesses this message first, then navigates to the message's receiver *gui*, its base component *GUI*, and finally the operation *init*(). The scope of constraint C1(*start*) is therefore the model elements $\{start, gui, GUI, init()\}$. A constraint instance needs to be re-evaluated if and only if elements in its scope changes.

This incremental inconsistency checker enables us to compute the constraint instances that are affected by changing a given location – those that have the location in their scope. As a result, changes made to a model only trigger re-evaluations of the affected constraint instances, rather than all the constraint instances. In addition, the scope of a constraint instance is also the basis for resolving a violation of the constraint (i.e. an inconsistency) since it indicates the locations that may need fixing. This incremental inconsistency checking approach has been shown empirically to be highly scalable for large, industrial UML models [8].

IV. Definitions

We provide here a few basic definitions. Firstly, we define an architectural model which represents a software system as below.

Definition 1. *An architectural model consists of a set of model elements, each of which is defined by a universally unique identifier (UUID) and a type (i.e. metaclass). A model element has a number of structural features, whose value can be a primitive type or a reference to other model elements.*

The definitions of model elements (i.e. their type and features) are described in detail in a metamodel (e.g. UML metamodel). For example, *Control* is a model element of type *Class* in the architectural model of our running example (see Figure 1). Component *Control* has a *name* feature (of type string) or an *ownedOperation* feature (a reference to a set of operations, which are model elements of type *Operation*, in the class). Component *Control* has an UUID and it is assumed that although a model element may be changed in various versions, its UUID does not change[2]. Change actions are formally defined as below.

Definition 2. *There are three possible types of primitive actions performed upon a model: add(e, t) – add a new model element type t with the UUID of e; delete(e, t) – delete an existing model element of type t with the UUID of e; and modify(e, f, v_o, v_n) – modify the value of feature f of e from v_o to v_n.*

[2]In practice, most tool support for models also provide and use unique identification for model elements. For instance, the standard textual encoding of UML models using XML Metadata Interchange (XMI) requires an unique XMI identifier for each model element.

For example, Alice creates message *rinse* to component instance *ctrl*, which consists of a sequence of primitive actions: adding a new message *rinse*, modifying its *receiveEvent* feature, and modifying the *represent* feature of Lifeline *ctrl*.

Definition 3. *Each action a has a reverse action $\overline{a}$ with the opposite effect. Specifically, the reverse action of each atomic action is given in the following table.*

***Action** a*	***Reverse action** $\overline{a}$*
add(e, t)	*delete(e, t)*
delete(e, t)	*add(e, t)*
modify(e, f, v_o, v_n)	*modify(e, f, v_n, v_o)*

Definition 4. *The difference Δ between two versions M_{old} and M_{new} of a model is a sequence of actions that when applied to model M_{old}, yields model M_{new}, i.e. $M_{old} + \Delta = M_{new}$. The reverse of Δ, denoting as $\overline{\Delta}$, is a sequence of actions that when applied to M_{new}, yields M_{old}, i.e. $M_{new} + \overline{\Delta} = M_{old}$.*

Note that $\overline{\Delta}$ is computed simply by reversing the sequence in Δ and replace each action with its reverse. When two (or more) difference sets of changes Δ_1 and Δ_2 (from two different versions) are applied to the same model (i.e. the common ancestor), conflicts may arise due to contradicting changes. Two typical scenarios of a conflict are when one software architect modifies a feature of a model element deleted by the other (i.e. $modify(e,f,v_o,v_n)$ in Δ_1 and $delete(e,t)$ in Δ_2), and when both software architects modify the same model element feature in different ways ($modify(e,f,v_o,v_n)$ in Δ_1 and $modify(e,f,v_o,v_{n'})$ in Δ_2). In our running example, Alice and Bob rename message *start* differently, which causes a conflict. Note that equivalent changes (e.g. creating a new component with the same name) may also be considered as a conflict but we deal with this simply by considering them as equal (i.e. the same UUID) and merging their features, i.e. a model element is included in the merged model which contains all features of both.

An (initial) merged model is created by applying the non-conflicting set of changes to the common ancestor model.

Definition 5. *Let Δ'_1 and Δ'_2 be the set of non-conflicting changes in the difference Δ_1 between model M_1 and M and the difference Δ_2 between model M_2 and M respectively. The (initial) merged model M_i is obtained by applying changes in Δ'_1 and Δ'_2 to M, i.e. $M_i = M + \Delta'_1 + \Delta'_2$. The set of available repair actions for resolving inconsistencies in M_i is $\Theta = \overline{\Delta'_1} \cup \overline{\Delta'_2} \cup (\Delta_1 - \Delta'_1) \cup (\Delta_2 - \Delta'_2)$.*

Figure 2 shows the set of available actions Θ for our running example. For example, in version 1 Alice has added an inheritance relationship from component *Driver* to component

$\overline{\Delta_1'}$ (The reverse of Alice's non-conflicting changes)	delete(Driver-inherit-Control, Generalization)
	delete(rinsing, State)
	delete(rinse, Message)
$\Delta_1 - \Delta_1'$	*<modify(Message[start], name, 'start', 'run'),*
$\Delta_2 - \Delta_2'$	*modify(Message[start], name, 'start', 'init')>*
$\overline{\Delta_2'}$ (The reverse of Bob's non-conflicting changes)	delete(Control-inherit-Driver, Generalization)
	modify(Operation[init], name, 'init', 'run')
	delete(stop, Message)
	delete(turnOff, Message)

Fig. 2. The set of available actions Θ for our running example.

Control, and thus this change action is part of the non-conflicting changes Δ_1' between version 1 and the original version. The reverse of this action, i.e. deleting the *Driver-inherit-Control* relationship (of type *Generalization*), is part of $\overline{\Delta_1'}$. Note that the conflicting actions involving renaming message *start* are stored as a pair (in *italic* in Figure 2).

Definition 6. *A repair plan P for an inconsistency I in model M is a minimal sequence of actions S when performed on M yields a new model M' and the inconsistency I is resolved in M'. Action sequence S is minimal in that removing any actions from it always results in a sequence that no longer resolves I in M.*

The reverse of repair plan *P*, denoting as $\overline{P}$, is obtained by reversing the sequence in *P* and replacing each action with its reverse.

V. Principle

The main objective our approach is providing a guidance mechanism to support the software architects in merging their concurrent changes to the model while preserving its consistency. Our approach is built atop existing model versioning technologies in order to use their capabilities in identifying the differences between versions of an architectural model (e.g. [1, 9, 10]), and obtaining an initial merged model in which non-conflicting changes are merged (see [2] for a review of existing model merging techniques). We guide the software architects to resolve inconsistencies found in the initial merging by a combination of three methods: (a) reversing the non-conflicting changes which have been applied; (b) applying a (non-conflicting) subsets of conflicting changes; and (c) making further "new" changes to the model.

Details of our merging process are described as below. We assume here a common ancestor model *M* that has two concurrent versions M_1 and M_2 that need to be merged.

1) First, we compute the difference Δ_1 between M_1 and M, and the difference Δ_2 between M_2 and M. We then identify conflicting changes and non-conflicting changes in Δ_1 and Δ_2.
2) Next, we create an initial merged model M_i by applying non-conflicting changes (in Δ_1 and Δ_2) to the common ancestor model M.
3) Based on the changes in Δ_1 and Δ_2, we establish the available action set Θ which contains the reverse of non-conflicting changes and pairs of conflicting changes (see Definition 5 in Section IV).
4) We then use our incremental consistency checker [8] to identify inconsistencies in the initial merged model M_i.
5) We explore to find which of those inconsistencies (and how they) can be fixed using change actions in Θ (*non-persistent* inconsistencies) and which of those cannot be fixed (*persistent* inconsistencies) using the available actions.

Since steps 1 – 4 can be done using existing model versioning techniques and inconsistency checking approaches, our focus in this paper is on step 5. In the next sections, we first present a naive and expensive approach to step 5, and then describe in detail our much more efficient approach which is also able to identify new changes that can be applied to the model to resolve persistent inconsistencies.

VI. Trial-and-Error Approach

The exploration to find out how non-persistent inconsistencies in the initial merged model M_i can be resolved using available actions in Θ (step 5 in Section V) can be done in a simple, brute-force manner. This approach would try applying all possible combinations of actions in Θ onto M_i to see which combination(s) resolves all the non-persistent inconsistencies and does not cause any new inconsistencies. The number of combinations that we need to iterate through is $2^{\#changes}$ where *#changes* is the number of change actions in Θ (number of *k*-combinations for all *k* from 1 to *#changes*). Thus, the computational complexity of such a brute-force approach is unfortunately exponential with the number of change actions in Θ, i.e. $O(\#C * 2^{\#changes})$ where $\#C$ is the total number of consistency constraints imposed on the model.

An improved version of this approach is to perform the exploration incrementally. Specifically, we consider one (non-persistent) inconsistency at a time and enumerate through only combinations of actions in Θ that may affect the truth value of the constraint associated with the inconsistency (i.e. accessing the constraint's scope). We then try applying each of those combinations of changes to the initial merged model. If the change does not resolve the inconsistency, we move on trying another combination. If the change actually resolves the inconsistency, we then continue a similar search to find available repairs for other non-persistent inconsistencies or new inconsistencies caused by the change until we found all possible combinations that can resolve all non-persistent inconsistencies in the initial merged model.

The improved approach utilizes the scope of a constraint (see the end of Section III) to reduce the number of combinations that need to be enumerated to identify fixes for the constraint violation: from $2^{\#changes}$ (as in the brute-force approach) to $2^{\#scopeChanges}$ where *scopeChanges* is the number of change actions in Θ that access the scope of the constraint. Nonetheless, it still involves iterating through an exponential number of combinations just to find those combinations that resolve an inconsistency. Thus, the worst-case computation complexity of this approach is still exponential with the number of change actions in Θ (where all actions in Θ access a constraint's scope). Another serious limitation of this approach is that it cannot suggest new changes (i.e. not available in Θ) to resolve persistent inconsistencies.

VII. Repair Generation Approach

Although the trial-and-error approach needs to enumerate through a large number of combinations of available actions, there is only a very small number of them which can resolve an inconsistency. In fact, previous work [5] has shown that independent of the model size, there are on average only 12 possible repairs per inconsistency[3] (see Figure 3). Thus, it is inefficient to enumerate through (e.g.) 2^{100} combinations, only to find 12 of which can actually resolve an inconsistency.

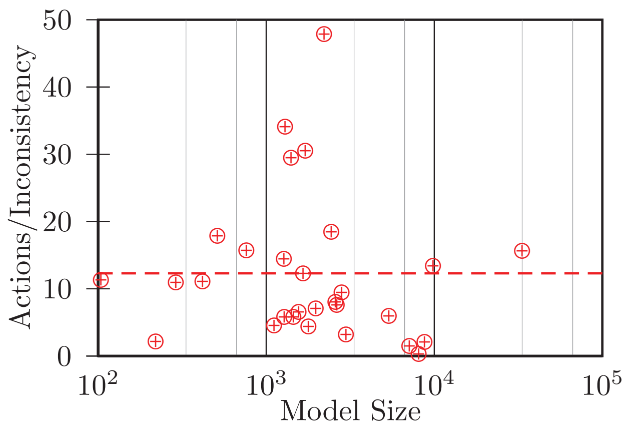


Fig. 3. Average Number of Repairs/Inconsistency

We propose here a more efficient approach which generates the exact repair(s) for each inconsistency by analyzing the structure of a consistency constraint and its expected and observed validation results (through observing the constraint's validation) to determine exactly which parts of the inconsistency must be repaired. In the following we briefly outline the approach presented in [5] and explain how this approach has been extended to be able to make repair actions concrete (i.e. a concrete value is known).

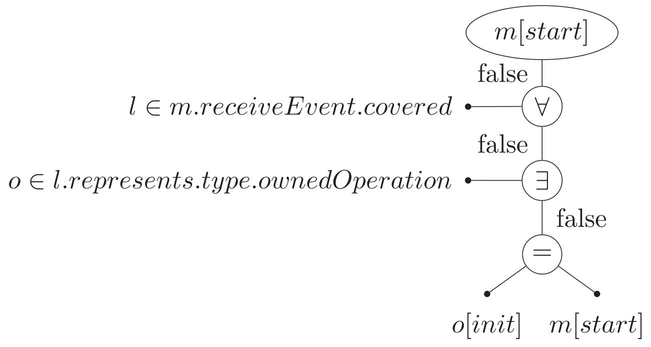
A. Repair Generator

The basis for the repair generation is the so called validation tree [5], which is created when a constraint instance is first evaluated. For illustration we formalize constraint C1 as below.

$$\begin{aligned}&\textit{Message } m:\\&\quad(\forall\, l \in m.receiveEvent.covered:\\&\qquad\exists\, o \in l.represents.type.ownedOperation:\\&\qquad\quad o.name = m.name)\end{aligned}$$

[3]Data collected from the evaluations on 29 industrial UML models and 18 consistency rules written in OCL.

This constraint is written for a context element of the type *Message*. The validation tree which was created when evaluating constraint C1(*start*) is shown in Figure 4.



$m[start]$
false
$l \in m.receiveEvent.covered$
$\forall$
false
$o \in l.represents.type.ownedOperation$
$\exists$
false
$=$
$o[init]$
$m[start]$

Fig. 4. Validation Tree for C1(*start*)

The validations starts at the model element *start* (the root node of the validation tree). The first operation executed is the universal quantifier ($\forall$) that iterates over all lifelines that the message is sent to (UML allows a message to be sent on more than one lifeline). The lifelines are accessed through the properties *receiveEvent* and *covered* from the message (m) *start*. The universal quantifier has as its condition an existential quantifier ($\exists$) that iterates over the operations of the component that is the type of the lifeline. This is done by accessing the properties *represents* (instance *gui* of the lifeline), *type* (component *GUI*), and *ownedOperation* (*init*). The condition of the existential quantifier compares ($=$) the message name (*start*) with the name of each operation (only *init* in this case). Since there does not exist an operation that is named *start*, the existential quantifier validates to *false* and thus the result of the complete constraint validation is also *false* (i.e. an inconsistency has been detected). More details of how a validation is built can be found in [5].

A repair tree is built based on the validation tree. The nodes of the repair tree are directly derived from the validation tree: $\forall, \wedge \rightarrow +$ (combinations of repair actions), and $\exists, \vee \rightarrow \bullet$ (alternative repair actions). Each branch of a validation tree has an expected and validated result. Note that in Figure 4 only the validated results are shown. The expected result for a constraint is always true and will be propagated top-down in the validation tree. A negation in the constraint will cause an inverting of the expected result (*true* $\leftrightarrow$ *false*). A mismatch between the expected and the validated result triggers the generation of repair actions. The type of the repair actions (i.e. create, delete, and modify) is derived from the logical operators and quantifier types: $\forall \rightarrow$ *delete*, $\exists \rightarrow$ *create*, and $= \rightarrow$ *modify*). The model elements that must be changed are the leaves of the logical expression that are violated (mismatch of the expected result to the validated result) in the validation tree.

However, repair actions generated in [5] are *abstract* repairs. In this example, one abstract repair generated (as in [5]) is the renaming of operation *init*, denoted as *modify(Operation[init], name)*, but it does *not* reveal which string to rename the operation to. We therefore extend the work in [5] to compute *concrete* repairs. The new repair generator takes additional information which is a set of available (concrete) actions Θ. Values for the repairs are derived from actions in Θ, i.e. Θ is used as the source for providing concrete values to instantiate the abstract repairs generated. For example, renaming operation *init* to *run* (i.e. *modify(Operation[init], name, 'init', 'run')*) is an action in Θ (see Figure 2) which is an instance of the abstract repair *modify(Operation[init], name)*. Another abstract repair suggested by the repair tree is the renaming of message *start* and *modify(Message[start], name, 'start', 'run')* is in Θ (i.e. an instance of the abstract repair). Therefore, another concrete repair for $\mathsf{C1}(\textit{start})$ is *modify(Operation[init], name, 'init', 'run')* and (+) *modify(Message[start], name, 'start', 'run')*. Figure 5 shows the full concrete repair tree for the validation shown in Figure 4 using the available actions in Θ. This repair tree represents two alternative, available repair plans for resolving inconsistency $\mathsf{C1}(\textit{start})$: renaming message *start* to *'init'*, or renaming both the message and operation *init* to *'run'*.

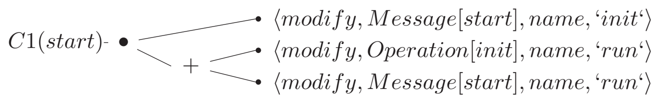

$C1(start)$-
$\langle modify, Message[start], name, \text{'}init\text{'}\rangle$
+
$\langle modify, Operation[init], name, \text{'}run\text{'}\rangle$
$\langle modify, Message[start], name, \text{'}run\text{'}\rangle$

Fig. 5. Concrete repair tree for $\mathsf{C1}(\textit{start})$

Algorithm 1: mergeExplore(): explore merging options

```
   Input: M_i, a model; fixedInconsistencies, a set of fixed
          inconsistencies; Θ, the available action set;
   Output: solutions, a set of alternative merged models
 1 begin
 2     inconsistencies := validate-incrementally(M_i)
 3     if inconsistencies.size() = 0 then
 4         solutions.add(M_i)
 5     else if inconsistencies ∩ fixedInconsistencies ≠ ∅
       then
 6         return ∅
 7     else
 8         I := inconsistencies.removeFirst()
 9         planList := get-available-repairs(M_i, Θ, I)
10         if planList.size() = 0 then
11             return ∅
12         else
13             fixedInconsistencies.add(I)
14             foreach P in planList do
15                 M'_i := execute-in-simulation(P, M_i)
16                 solutions.addAll(mergeExplore(M'_i, Θ,
                   fixedInconsistencies))
17                 M_i := execute-in-simulation(P̄, M'_i)
18     return solutions
   /* Initial call is mergeExplore(M_i, Θ, ∅)
      where M_i is the initial merged
      model.                                   */
```

B. Exploring Inconsistency Resolutions

The process of exploring options to resolve non-persistent inconsistencies[4] in an initial merged model M_i is described in Algorithm 1. First, the model is *incrementally* validated against a number of consistency constraints to identify a set of inconsistencies. If there is no (non-persistent) inconsistencies in the model, this model is a solution. Otherwise, we will explore incrementally how to fix them (one at a time) as follows.

We get the first inconsistency I (line 8 of Algorithm 1), and invoke the repair generator (described in the previous section) by calling function $get-available-repairs(M_i, \Theta, I)$ to retrieve a set of alternative repair plans for inconsistency I. Note that those repair plans are derived from the actions in the set of available actions Θ.

If there is no available repair plan found for inconsistency I, we terminate our exploration (lines 10 – 11). Otherwise, for each repair plan P (lines 14 – 17), we execute it (in simulation) by performing its actions onto model M_i, which yields model M_i'. Repair plan P may has positive side-effects (resolving some other inconsistencies) and/or negative side-effects (causing some new inconsistencies) or no side-effect. Therefore, we need to explore if M_i' contains any inconsistencies (either the same existing inconsistencies as in M_i or new inconsistencies caused by the application of repair plan P) and how to resolve them. This exploration continues recursively in a depth-first-search manner (line 16). After we have done with P, we will undo all of its actions to obtain back M_i (to avoid keeping multiple copies of the model) and do a similar exploration with the next alternative repair plan in the list.

Our algorithm also has a loop detection (lines 5 – 6) which keeps track of all inconsistencies that have been fixed so far in an exploration path (i.e. *fixedInconsistencies*) and checks if the same inconsistency is seen again in that path. In such cases, we have a loop and the exploration terminates with no solution and moves on to the next alternative. We also note that the algorithm tries to find *all* solutions (and present them to the software architect for selection), i.e. continue searching for other solutions even after a solution is found.

Figure 6 shows an example of how we explore (using Algorithm 1) to find alternative resolutions for non-persistent inconsistencies in the initial merged model in our running example. First, we would like to fix inconsistency $\mathsf{C1}(\textit{turnOff})$ and the repair generator gives us only one available option in Θ (see Figure 2) which involves the deletion of message *turnOff*. Applying this repair gives us a new model M_i^1 in which inconsistency $\mathsf{C1}(\textit{turnOff})$ has been resolved but inconsistencies $\mathsf{C3}(\textit{Control})$, $\mathsf{C3}(\textit{Driver})$ and $\mathsf{C1}(\textit{start})$ still

[4]Note that persistent inconsistencies are ignored in this exploration and will be dealt separately.

exist. We now want to fix $\mathsf{C3}(Control)$ and the repair generator gives us two available alternative repair plans: deleting *Driver-inherit-Control* or deleting *Control-inherit-Driver* relationship. We try applying the first repair which has a positive side-effect (also resolving inconsistency $\mathsf{C3}(Driver)$), yielding model M_i^2 which has only one inconsistency $\mathsf{C1}(start)$. The repair generator then gives us two alternative repair plans: renaming message *start* to *init*, and renaming message *start* to run() and renaming operation *init()* to *run()*. The outcomes of both options are consistent models (i.e. models M_i^3 and M_i^4), and thus they are part of the set of solutions presented to Alice and Bob.

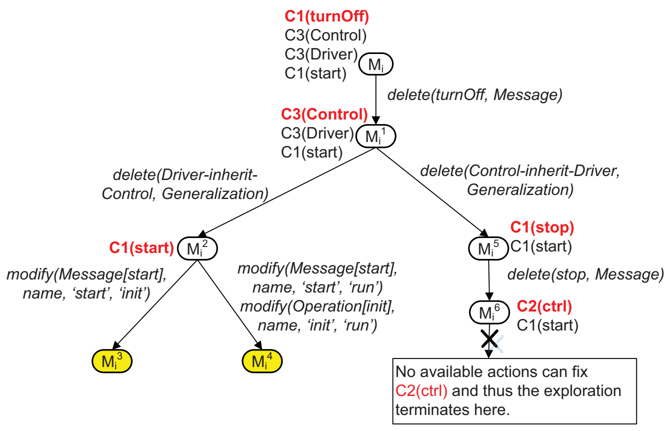

C1(turnOff)
C3(Control)
C3(Driver)
C1(start)
Mi
delete(turnOff, Message)
C3(Control)
C3(Driver)
C1(start)
Mi1
delete(Driver-inherit-Control, Generalization)
delete(Control-inherit-Driver, Generalization)
C1(start)
Mi2
modify(Message[start], name, 'start', 'init')
modify(Message[start], name, 'start', 'run')
modify(Operation[init], name, 'init', 'run')
Mi3
Mi4
C1(stop)
C1(start)
Mi5
delete(stop, Message)
Mi6
C2(ctrl)
C1(start)
No available actions can fix C2(ctrl) and thus the exploration terminates here.

Fig. 6. An example of how to find solutions for resolving all non-persistent incosnistencies

We then try the other repair plan for resolving inconsistency $\mathsf{C3}(Control)$, i.e. deleting *Control-inherit-Driver* relationship. This repair has one positive side-effect (also resolving inconsistency $\mathsf{C3}(Driver)$) and one negative side-effect (causing new inconsistency $\mathsf{C1}(stop)$ since component *Control* no longer inherits operation *stop()* from component *Driver*). The repair generator gives us one available repair to resolve inconsistency $\mathsf{C1}(stop)$, which is deleting message *stop*. We apply this repair (yielding model M_i^6) but it has a negative effect (causing new inconsistency $\mathsf{C2}(ctrl)$ since the transitions in the state machine diagram now do not match with the sequence of messages). We ask the repair generator but there is no available repair that can be formed for resolving $\mathsf{C2}(ctrl)$ and thus the exploration terminates here.

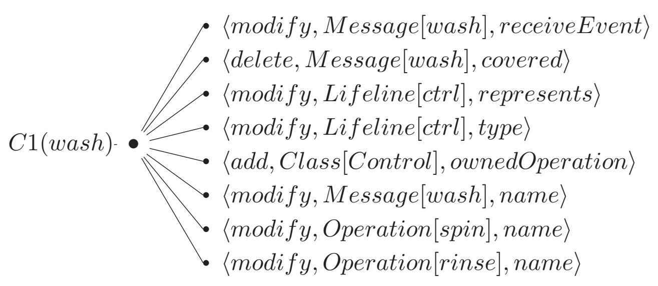

$C1(wash)$
$\langle modify, Message[wash], receiveEvent\rangle$
$\langle delete, Message[wash], covered\rangle$
$\langle modify, Lifeline[ctrl], represents\rangle$
$\langle modify, Lifeline[ctrl], type\rangle$
$\langle add, Class[Control], ownedOperation\rangle$
$\langle modify, Message[wash], name\rangle$
$\langle modify, Operation[spin], name\rangle$
$\langle modify, Operation[rinse], name\rangle$

Fig. 7. Abstract repair tree for $\mathsf{C1}(wash)$

Our repair generation approach is also able to identify new changes (i.e. not made in the versions) that can be applied to the model to resolve persistent inconsistencies and the potential side-effects of such changes. Specifically, for persistent inconsistencies (which we already know that we cannot resolve them using available actions in Θ), we will provide the software architects with a set of abstract repair plans represented in a hierarchical manner. For example, the set of abstract repair plans for resolving (persistent) inconsistency $\mathsf{C1}(wash)$ is shown in Figure 7, which suggests a number of ways to resolve this inconsistency such as modifying the message's name or the name of an existing operation.

VIII. Evaluation

A prototype implementation of our approach has been implemented and integrated with IBM Rational Software Architect (RSA) [11] and the Model/Analyzer consistency checker [12] (an implementation of [8]). The consistency constraints are written in the Object Constraint Language (OCL). We use the model differencing functionality provided with RSA to obtain the difference between model versions (and from that we derive the set of available actions Θ) and an initial merged model. The Model/Analyzer provides instant consistency checking on the initial merged model. The tool returns to the user a set of options that can resolve inconsistencies having arisen during merging. We now discuss an evaluation of our approach using the implemented tool.

A. Scalability

The computation time of Algorithm 1 basically depends on two factors: the number of non-persistent inconsistencies in the initial merged model (since a solution is found only when all non-persistent consistencies are resolved – see line 3 of Algorithm 1) and the average number of concrete repair plans found for an inconsistency (since the search for solutions tries each of those repair plans – see line 14 of Algorithm 1). The number of concrete repair plans in turn depends on the size of the model and the size of the available action set Θ (see line 9 of Algorithm 1). Therefore, the three scalability drivers of our approach are the size[5] of Θ, the model size and the number of non-persistent inconsistencies in the initial merged model.

We have evaluated our approach on four industrial UML models of different sizes: 290, 2,212, 16,255 and 33,347 model elements. The evaluations were done on 64bit Linux (3.7) with Intel Core 2 Quad CPU (Q9550) @ 2.84Ghz, 8GB RAM (4GB available for the IBM RSA). We used 18 consistency rules written in OCL (all of them are described in [13]) which produce from 92 to 13,504 constraint instances in those models. Since we did not have available large UML models with multiple versions, for the purpose of a scalability assessment, we have created the versions of the models by randomly introducing a number of changes to each model. For each set of changes, an initial merged model and a set of

[5]Note that the size of Θ is actually the number of changes in the model versions to be merged.

available actions Θ were created and they were the input to our tool. We have also tested our approach with different sizes of Θ, i.e. from containing only 3 up to 1,650 change actions. We then measured the time our tool took for finding all possible solutions for resolving all non-persistent inconsistencies found in the initial merged model.

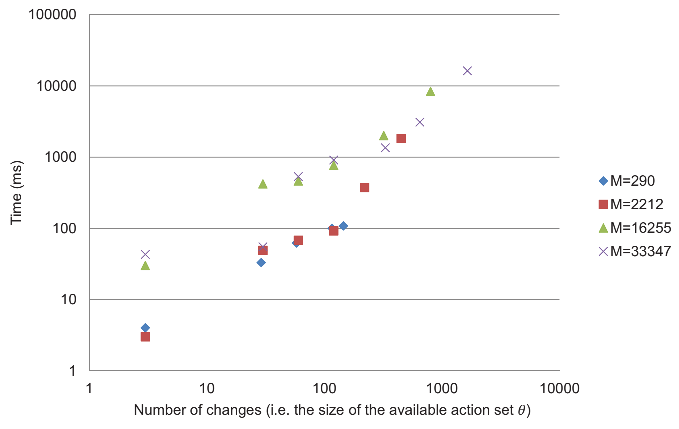


Fig. 8. Computation time for different models and changes

Figure 8 shows the computing time for all four models against the number of change actions in Θ (noting that both graphs in Figures 8 and 9 are in the logarithmic scale). The first important observation in this result is that the computing time increases *linearly* with the number of change actions in Θ across all four models. This result confirms the superior efficiency of our repair generation approach, compared to the trial-and-error approach where the time taken grows exponentially with the number of changes. In addition, this result demonstrates that the computing time does not increase as the model size increases (e.g. with 30 change actions in Θ, it took approximately the same 50ms for the smallest model with 290 model elements and the largest model with 33,357 model elements). Therefore, our approach can scale to very large models and to very large numbers of changes in the model versions to be merged. For example, with the model of 33,347 elements and 1,650 changes to be merged, it took our tool less than 17s to find all 9 possible solutions for resolving all 71 inconsistencies in the initial merged model.

Figure 9 shows the computing time against the number of non-persistent inconsistencies in the initial merged model. As can be seen, the time taken to find all the solutions to resolve those inconsistencies grows polynomially with their number. Our approach can also quickly find solutions to resolve a large number of inconsistencies: it took only less than 9s for finding all the 6 possible solutions which resolve all 100 inconsistencies in the model with 16,255 model elements. We note that the larger the number of solutions, the longer it takes to find all of them. For example, in the case of the largest model with 33,347 model elements, there were 65 solutions for fixing one inconsistency (which took 532ms to find them all) whereas there were only 11 solutions for fixing two inconsistencies (which took 55ms).

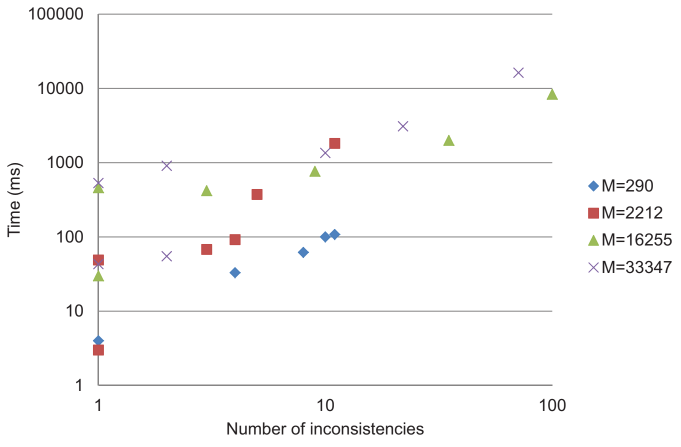


Fig. 9. Computation time for different models and inconsistencies

B. Correctness

Another important aspect for the evaluation is the correctness of our approach in terms of: (a) proposing only solutions that resolve all non-persistent inconsistencies and that are derived from actions in the available action set Θ (soundness); and (b) finding all of those solutions (completeness). We evaluate this by conducting the following process. First, we randomly remove some of the change actions in Θ. Doing so may prevent some/all of the non-persistent inconsistencies from being resolved since there may no longer exist in Θ the repair actions which are able to resolve them. The test is whether our approach could be used to correctly identify those randomly removed actions. This way we can test whether the guidance is giving correct responses in terms of resolving inconsistencies. The test is implemented and included with our scalability evaluation. The correctness test passed in all 29 scenarios (across the four models and different number of changes), which confirms that our approach is correct. Future work involves formally proving the correctness of our approach.

IX. Related Work

There have been a range of techniques and tools proposed for differencing and merging architectural models. For example, the work in [1] focuses on identifying the changes between versions of a product line architecture and merging those changes to create a consolidated version. The approach in [10] is for differencing and merging generic architectural models that follow the traditional component-and-connector (C&C) view. However, they only address structural models and do not deal with inconsistencies during the merging process. Those techniques are part of the large literature on model merging (see [2] for a recent survey and the online bibliography compiling an extensive list of relevant publications in this field [14]), on which our approach leverages to compute the differences between model versions and create an initial merged model (steps 1 & 2 in section V).

Existing model merging techniques mostly focus on dealing with conflicts. Some recent work start dealing with inconsistencies in model merging but they only focus on detecting inconsistencies (e.g. [3, 4]). Recently, the approach proposed in [15] tackles inconsistency resolution in model merging but it can only suggest highly abstract repairs (based on graph modification) to the user (as opposed to concrete repairs in our approach). The recent Eclipse's EMF Diff/Merge incubation project [16] also aims to support consistent merging of EMF models by computing the minimal superset of differences that must be merged to preserve consistency. Persistent inconsistencies, if existing, would cause a problem in their approach since they would invalidate any possible merge.

Automation of resolving inconsistencies in models has received increasing attention in recent years. Nentwich et al. [17] proposed an approach for automatically generating *abstract* repair options by analyzing consistency rules expressed in first order logic and models expressed in xlinkit. However, they did not take into account dependencies among inconsistencies and potential interactions between repair actions for fixing them. Some recent work (e.g. [5, 18, 19]) overcomes this limitation by analyzing the side-effects of the generated repairs. Nonetheless, such approaches can only generate abstract repairs and let the user work out the concrete repair actions. Our work in this paper tackles this issue by automatically generating concrete repairs from a set of available actions. Existing approaches also propose to resolve all inconsistencies at the same time. Those approaches however only scale up to medium-size models (e.g. [20]), or limits the depth of the search tree (e.g. [21]) against the favor of fixing all inconsistencies.

X. Conclusions and Future Work

This paper presented a novel approach for resolving syntactical and semantic inconsistencies in the merging of architectural model versions. Our approach is able to find all possible solutions which resolve all non-persistent inconsistencies introduced by merging different versions of an architectural model. Our approach also provides guidance for resolving persistent inconsistencies (which pre-exist in the model) in terms of telling the software architects exactly which model elements should be changed to resolve them. We have demonstrated through a number of empirical studies that our approach is scalable and not affected by the model size. More importantly, our approach scales very well to large numbers of changes in the versions to be merged, indicating its usefulness and efficiency in situations like merging branches where the difference between versions tends to be large. In terms of future work, we will evaluate our approach with real model versions from a variety of domains to confirm our findings (including the correctness of our approach) as in the current evaluation. In addition, we will perform an evaluation of our tool with human users to fully assess its effectiveness and usability.

References

[1] P. Chen, M. Critchlow, A. Garg, C. Westhuizen, and A. Hoek, "Differencing and merging within an evolving product line architecture," in *Software Product-Family Engineering*, ser. Lecture Notes in Computer Science, F. Linden, Ed., 2004, vol. 3014.

[2] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer, "An introduction to model versioning," in *Formal Methods for Model-Driven Engineering*, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds. LNCS 7320: Springer, 2012, pp. 336–398.

[3] P. Brosch, U. Egly, S. Gabmeyer, G. Kappel, M. Seidl, H. Tompits, M. Widl, and M. Wimmer, "Towards semantics-aware merge support in optimistic model versioning," in *Models in Software Engineering - Workshops and Symposia at MODELS 2011, Wellington, New Zealand, October 16-21, 2011, Reports and Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 7167. Springer, 2012, pp. 246–256.

[4] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chećhik, "Global consistency checking of distributed models with TReMer+," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08, 2008, pp. 815–818.

[5] A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012, 2012, pp. 220–229.

[6] J. T. Lallchandani and R. Mall, "A dynamic slicing technique for UML architectural models," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 737–771, Nov. 2011.

[7] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J. R. O. Silva, "Documenting component and connector views with UML 2.0," Software Engineering Institute (Carnegie Mellon University), Tech. Rep. CMU/SEI-2004-TR-008, 2004.

[8] A. Egyed, "Instant consistency checking for the UML," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, 2006, pp. 381–390.

[9] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, ser. ASE '05, 2005, pp. 54–65.

[10] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan, "Differencing and merging of architectural views," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06, 2006, pp. 47–58.

[11] IBM, "IBM Rational Software Architect," http://www.ibm.com/software/rational/products/swarchitect/, Accessed 26 September 2013.

[12] A. Reder and A. Egyed, "Model/Analyzer: a tool for detecting, visualizing and fixing design errors in UML," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10, 2010, pp. 347–348.

[13] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 188–204, March 2011.

[14] "Bibliography on Comparison and Versioning of Software Models," http://pi.informatik.uni-siegen.de/CVSM/, Accessed 26 September 2013.

[15] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer, "A fundamental approach to model versioning based on graph modifications: from theory to implementation," *Software and Systems Modeling*, 2012, Online First.

[16] Eclipse, "EMF Diff/Merge," http://www.eclipse.org/diffmerge/, Accessed 26 September 2013.

[17] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 455–464.

[18] H. K. Dam and M. Winikoff, "Supporting change propagation in UML models," in *Proceedings of the 26th IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[19] H. K. Dam, L.-S. Le, and A. Ghose, "Supporting change propagation in the evolution of enterprise architectures," in *Proceedings of the 2010 14th IEEE International Enterprise Distributed Object Computing Conference*, ser. EDOC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 24–33.

[20] H. K. Dam and M. Winikoff, "An agent-oriented approach to change propagation in software maintenance," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 23, no. 3, pp. 384–452, 2011.

[21] M. A. A. da Silva, A. Mougenot, X. Blanc, and R. Bendraou, "Towards automated inconsistency handling in design models," in *Proceedings of the 22nd international conference on Advanced information systems engineering*, ser. CAiSE'10, 2010, pp. 348–362.