The Journal of Systems and Software 112 (2016) 137–155

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Consistent merging of model versions

Hoa Khanh Dam[a,*], Alexander Egyed[b], Michael Winikoff[c], Alexander Reder[b], Roberto E. Lopez-Herrejon[b]

[a] University of Wollongong, Australia
[b] Johannes Kepler University, Austria
[c] University of Otago, New Zealand

ARTICLE INFO

Article history:
Received 31 July 2014
Revised 23 April 2015
Accepted 17 June 2015
Available online 27 June 2015

Keywords:
Model merging
Inconsistency management
Model versioning

ABSTRACT

While many engineering tasks can, and should be, manageable independently, it does place a great burden on explicit collaboration needs—including the need for frequent and incremental merging of artifacts that software engineers manipulate using these tools. State-of-the-art merging techniques are often limited to textual artifacts (e.g., source code) and they are unable to discover and resolve complex merging issues beyond simple conflicts. This work focuses on the merging of models where we consider not only conflicts but also arbitrary syntactic and semantic consistency issues. Consistent artifacts are merged fully automatically and only inconsistent/conflicting artifacts are brought to the users' attention, together with a systematic proposal of how to resolve them. Our approach is neutral with regard to who made the changes and hence reduces the bias caused by any individual engineer's limited point of view. Our approach also applies to arbitrary design or models, provided that they follow a well-defined metamodel with explicit constraints—the norm nowadays. The extensive empirical evaluation suggests that our approach scales to practical settings.

1. Introduction

Models have become central artifacts which are created and used by software engineers. In a collaborative environment, which is the dominant form of today's software development, software engineers concurrently and independently work on models which subsequently need to be merged. A basic scenario is where multiple software engineers work independently on a single 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 engineers. 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 engineers find themselves confronted with concurrently evolving versions of architectural models (Chen et al., 2004). All these scenarios pose the challenging need to merge these different versions of 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 (Brosch et al., 2012b). 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. Brosch et al., 2012a; Sabetzadeh et al., 2008) and there has been very little work in resolving such inconsistencies having arisen during model merging.

This paper contributes a novel approach to model merging which helps software engineers in combining versions of models that are created and maintained separately. Our approach considers arbitrary, user-definable consistency constraints and merges the model versions fully automatically if they are consistent and free of conflicts.[1] The software engineers are notified only if there are

* Corresponding author. Tel.: +61 242214875.
E-mail addresses: hoa@uow.edu.au (H.K. Dam), alexander.egyed@jku.at (A. Egyed), michael.winikoff@otago.ac.nz (M. Winikoff), alexander.reder@jku.at (A. Reder), roberto.lopez@jku.at (R.E. Lopez-Herrejon).
URL: http://www.uow.edu.au/~hoa/ (H.K. Dam), http://www.alexander-egyed.com (A. Egyed), http://infosci.otago.ac.nz/michael-winikoff (M. Winikoff)

[1] Two models are in *conflict* if a model element is changed differently in each of the models, for instance if it is modified in one and deleted in another. The models are *inconsistent* if desired constraints do not hold in the merged model.

http://dx.doi.org/10.1016/j.jss.2015.06.044
0164-1212/© 2015 Elsevier Inc. All rights reserved.

conflicts or inconsistencies. However, since inconsistencies are more complex problems than simple conflicts, solving them becomes harder. Repairing an inconsistency can have the side effect of creating a different inconsistency ("cascading"). Furthermore, the number of alternative repairs increases exponentially with the complexity of the consistency rule and the number of elements accessed (Reder and Egyed, 2012). Previous work has shown that abstract repairs, which merely identify the model elements that require repairing, are reasonably localized and scalable to compute. On the other hand, 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 infeasible endeavor.

This paper is a substantially extended and revised version of Dam et al. (2014) in a number of aspects. We have improved and extended our merging algorithm to include pruning (in the search) and catering for conflicting actions (Section 5). In addition, the new merging algorithm utilizes the scope of a consistency constraint (Egyed, 2006) in deriving candidate merged models. This approach offers an alternative to using the repair generation as in the previous version (Dam et al., 2014). Another significant extension is the formal proof which establishes the correctness of our approach (Section 6.1). The evaluation was also extended to accommodate the new merging algorithm. Sections 1 and 2 are also extended to better motivate and articulate the model merging problem, while Section 7 is extended to provide a more comprehensive review of the literature.

In this paper, we argue that the space of repairs for resolving inconsistencies in model merging is constrained by the changes made to the original model and thus it is practically feasible to explore them—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). If there are conflicts and/or inconsistencies among the artifacts to be merged, then clearly a compromise between those artifacts is necessary. A repair in this sense reflects a compromise. The constrained search space implies that there are limited resolution opportunities, and our approach employs a fast, automated search technique to quickly gauge whether a compromise is possible to solve the merging problem by taking some (but not all) of the engineers' changes. It is useful to automate this initial compromise to avoid bias. However, since merging may involve tradeoffs where human judgment and communication are required, our approach provides the software engineers with all feasible alternative compromises in order to help them make informed, consistent merging decisions. The benefits of our approach are:

1. Artifacts are merged fully automatically if they are consistent and conflict-free.
2. Inconsistencies and/or conflicts caused during the merging are instantly recognized and reported to the engineer.
3. Even with inconsistencies, parts of artifacts are still merged automatically if they are not involved in the inconsistencies.
4. Unbiased compromises for resolving the inconsistencies among the engineers' artifacts are computed automatically, to help the engineers quickly assess the problem.

We believe that our approach is applicable to arbitrary modeling languages and software engineering artifacts, as long as they follow a well-defined metamodel with explicit constraints. Since today the standard Unified Modeling Language (UML) is predominantly used in the industry for representing software models (Malavolta et al., 2013), we illustrate and validate our work mostly in the context of UML models. Architectural description languages such as Architecture Analysis and Design Language (AADL) (Feiler et al., 2006) have a metamodel, and constraints such as "A process can only be a subcomponent of a system component" can be expressed upon the metamodel. Temporal constraints modeling component interaction as expressed in the AADL's behavior annex may need special treatments but our technique still can apply in general. We demonstrate that our approach is correct and an empirical analysis of large, third-party, industrial software models indicates its computational efficiency and scalability in practice. We do not presume the original model (or the versions) to be fully consistent, nor is there an expectation that the final, merged model must be consistent. This approach can thus be used at any level of maturity of the model – and hence at any stage of the process – to support the collaborative merging of artifacts.

Table 1
Example of consistency constraints.

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

[a] 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.

The structure of our paper is as follows. In the next section, we will describe a typical scenario in which the key limitations of existing model merging techniques are highlighted. We then discuss how inconsistencies occur in merging models in Section 3. Section 4 serves to describe an architectural overview of our approach and its details are provided in Section 5. We then prove the correctness of our approach and report a number of experiments to validate its scalability in Section 6. Finally, we discuss related work in Section 7 before we conclude and outline future work in Section 8.

2. Illustrative example

We describe here a typical example of classical model merging where two software engineers, Alice and Bob, concurrently work on developing a model for a software controlling a washing machine. In this example, Alice and Bob use the Unified Modeling Language (UML) which has extensively been used for representing the models of software systems in recent years (Ivers et al., 2004; Lallchandani and Mall, 2011; Malavolta et al., 2013). We however note that our approach also applies to arbitrary models as long as they follow a well-defined metamodel with explicit consistency constraints, which is today's norm. Such constraints specify the required syntactical (e.g. well-formedness) and semantical consistency (e.g. coherence between different views) for a model. Table 1 describes three typical consistency constraints on how a UML sequence diagram relates to class and state machine diagrams and the inheritance relationship between classes in the class diagram. These three constraints are taken from the literature (C1 and C2 from Egyed, 2006) and UML specifications (C3).

Fig. 1 shows a UML fragment of the model which covers both the structural view (a class diagram) and the behavioral views (a sequence diagram and a state diagram). Alice's class diagram describes three classes *GUI*, *Control* and *Driver* and their relationships: 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 classes *GUI* and *Control*, whereas the state machine diagram shows the behavior of the controller of the washing machine, i.e. class *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 (see version 1 in Fig. 2) designs behavioral aspect of the new rinsing feature by adding message *rinse*

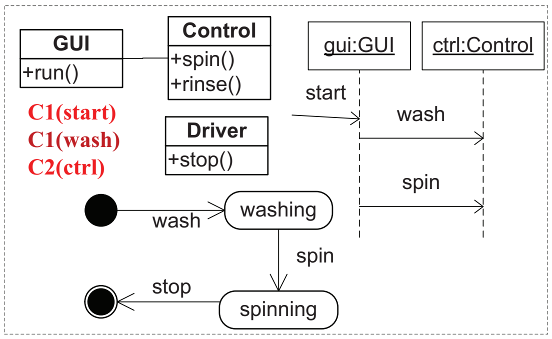


Fig. 1. Original version (the common ancestor).

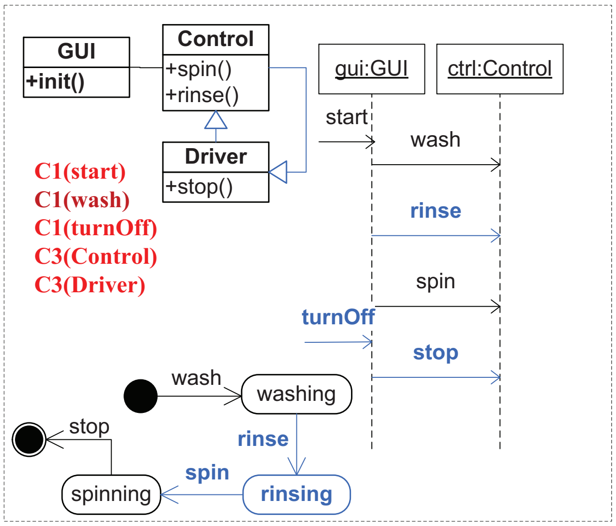


Fig. 3. The merged version.

to object *ctrl* (R5 in Fig. 2) and adding state *rinsing* and its associated transitions in the state machine diagram (R2, R3, and R4). She also notices that the model has an inconsistency, i.e. C1(*start*): the sequence diagram has a message named *start* to object *gui*, but class *GUI* does not have a corresponding operation. She therefore renames message *start* to *run* (R6). She also makes class *Driver* become a subtype of class *Control* (R1), possibly thinking that a driver should be a special control.

In the meanwhile, being unaware of Alice's changes, Bob (see version 2 in Fig. 2) completes the design for the stopping feature by making class *Control* become a subtype of class *Driver* (R8), adding a message *stop* to object *ctrl* (R10), and adding a message *turnOff* (sent to object *gui*; R11). He also attempts to fix the inconsistency C1(*start*) by renaming both operation *run()* to *init()* (R9) and message *start* to *init* (R7). Although both Alice and Bob's models solve the same inconsistency C1(*start*), they do so in a manner that is not compatible. When both engineers check in their own version, the merging process should recognize this problem, and this is their first real chance to collaborate with the goal to identify conflicts and inconsistencies among their work (if any), and identify possible solutions. This example demonstrates the three-way merging approach where the merging considers the two models as well as their common ancestor. The output is produced by applying this three-way model merging approach where the merging considers the two model versions as well as their common ancestor. The three-way merging approach is widely used in almost all versioning systems (including text, code and model versioning systems) (Brosch et al., 2012b; Mens, 2002). Most of existing model versioning systems would typically produce a merged version as in Fig. 3 and highlight a conflicting change: both Alice and Bob renamed message *start* differently. The system then would ask (either of) them to deal with the conflict. However, doing so only addresses conflicts, not inconsistencies.

Even if the direct conflict (the incompatible names) is resolved, the resulting model still has inconsistencies. Firstly, since Alice and Bob have each created an inheritance relationship between classes *Control* and *Driver*, but in different directions, both of them are integrated into the merged version which now has an illegal circular inheritance (violating constraint C3). Secondly, in the merged version the new message *turnOff* does not match with any operation in class *GUI*, which causes another inconsistency (i.e. violation of constraint C1*(turnOff)*). In addition, there is still no operation in class *Control* matching with message *wash*, and thus inconsistency C1(*wash*) still exists in the merged version. Finally, constraint C1*(start)* is violated in the original version (since message *start* received by instance *gui* of class *GUI* does not match with any operation in the class), 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 manual resolution. Hence, the choice made to resolve the conflict (i.e. Alice's or Bob's renaming of *start*()) has consequences on the resolution of inconsistencies.

It is also noted that Bob and Alice both also made changes that are not related to the inconsistencies above. For example, Alice added the state *rinsing* in the state diagram, and it is easy to see that this change is neither related to the circular inheritance problem nor to the conflicting renaming of the method/message. Therefore, it is important to: (a) recognize automatically that the added state is independent of the other changes; and (b) preserve the added state even though there are inconsistencies. It is also noted that in the case where designers make changes to separate parts of a model, inconsistency (between those parts) may also arise, and our approach is also able to deal with them.

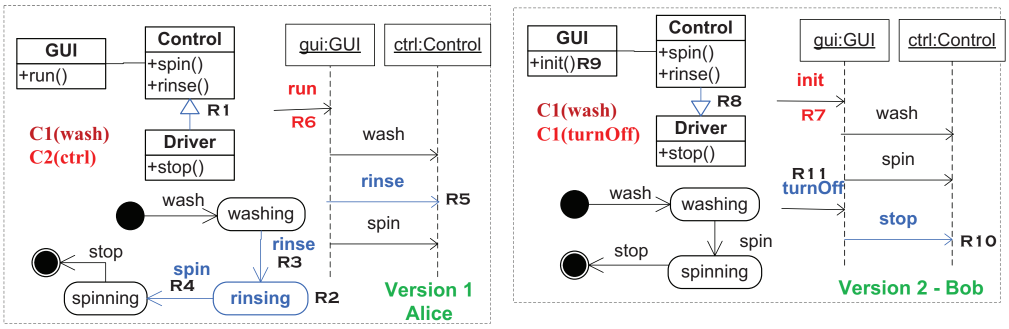


Fig. 2. Alice's and Bob's versions.

Table 2
Life-cycle of an inconsistency: being present (P) or absent (A). 1, 2, 3, and 4 denote Patterns 1, 2, 3, and 4 respectively.

Model	1	2			3			4	No inconsistency in merged							
Original	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
Merged	P	P	P	P	P	P	P	P	A	A	A	A	A	A	A	A

Model merging needs to preserve consistent merges, report inconsistencies, and explore alternative merging solutions for inconsistencies (attempting to maximize consistency in the merged model while preserving as many changes made by Bob and Alice as possible). In the rest of the paper, we will show that our approach is able to identify whether a merge is possible and if not which compromises are necessary. The important contribution here is that our approach does not do that on the granularity of an entire model but rather at the granularity of individual consistency checks. For example, Bob's addition of the inheritance relationship is not related to the conflicting renaming of the method. Our approach recognizes that the merging of the one is possible even if the merging of the other is not. In addition, it recognizes that there are choices for merging the others which are presented to Alice or Bob.

3. Lifecycle of 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 2 captures a typical lifecycle of an inconsistency in terms of its *presence* (denoted as P) and *absence* (denoted as A) in a given version of a model. There are four model versions that we are interested in here: the original model (i.e. the common ancestor), the versions to be merged (i.e. Versions 1 and 2), and the merged model. The presence of an inconsistency in a model version means that the corresponding consistency constraint has been instantiated and evaluated as being inconsistent in that model. By contrast, the absence of an inconsistency indicates that the related constraint instance either has been evaluated as being consistent or the constraint instance no longer exists due to the deletion of the context element. Table 2 captures all possible scenarios (i.e. all permutations) of the presence and absence of an inconsistency in the models to be merged, the common ancestor and the merged model.

Inconsistencies existing in the original model may disappear in the merged model (e.g. C2(*ctrl*) in the running example, see Fig. 1) since the revised changes and/or the merging itself may have fixed them (see column "No inconsistency in merged" in Table 2). The merged model may however contain inconsistencies due to one of the following *patterns* of reasons:

1. Pattern 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 C1(*wash*) is an example of this inconsistency type.
2. Pattern 2: An inconsistency exists in the original model (e.g. C1(*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. Pattern 3: An inconsistency (e.g. C1(*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. Pattern 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 C3(*Control*) and C3(*Driver*) is an example of this inconsistency type.

Inconsistencies whose lifecycle follow the first pattern have preexisted in the original model and also existed in the versions. We will not be able to resolve such inconsistencies by reversing the revisions' changes since such changes were not the causes of the inconsistencies. Applying conflicting changes, if they exist, may be able to resolve such inconsistencies. If none of the changes can resolve an inconsistency, it is classified as *persistent inconsistency*.

Inconsistencies whose lifecycle follows 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. For example, an inconsistency following pattern 2 exists in the original model, but is absent in one or both revised versions (since either of the changes has fixed it) and then returns in the merged model (since merging the changes has re-created the inconsistency). Thus, reversing the changes (in one of the revised versions) that undo the fixing of the inconsistency will resolve the inconsistency. We refer to those inconsistencies as *non-persistent inconsistencies*.

Our approach detects inconsistencies using an existing incremental inconsistency checker (Egyed, 2006) which identifies model elements that are changed and that affect the truth values of consistency constraint instances. Such elements form the *scope* of a constraint instance, which is established by automatically observing which model elements are accessed during the evaluation of consistency constraints. This incremental inconsistency checker enables us to identify the constraint instances that are affected by changing a given location, which is a model element's field which affects the truth value of the constraint instances. 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 (Egyed, 2006).

4. Architectural overview

This section introduces the architecture of our merging framework (Fig. 4). The main objective of our approach is to provide a guidance mechanism to support software engineers in merging their concurrent changes to the model while maximizing its consistency. As input, our approach requires *versions*[2] of design models to be merged (2...*n*), their common ancestor (i.e., original model) to compute the changes made relative to the earlier state, and a customizable set of *consistency constraints* that consistent model merging should consider. The first phase of our merging process employs the existing

[2] The example and algorithm that we present here follows the three-way merging but note that our approach can be generalized to *n*-way model merging.

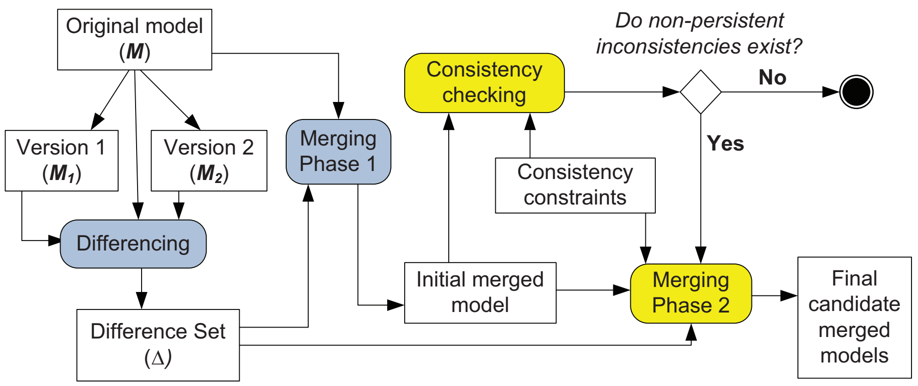


Fig. 4. A consistent merging framework for model versions.

three-way merging approach to obtain the initial merged model. We then check for non-persistent inconsistencies in the initial merged model, and if they do not exist, the process is finished and the initial merged model is the final outcome of the merging. If non-persistent inconsistencies are found, the process moves to the second phase where variations of the initial merged model are computed (by reversing or applying changes in the versions) to find consistent solutions. We now describe the merging process in more detail.

An architectural model, which represents a software system, is defined as below. Although we focus on UML models in this paper, the notions and ideas in our approach are generally applicable and customizable to other types of modelling languages.

Definition 1 (Software model)**.** A *software model* consists of a set of model elements.

Definition 2 (Model element)**.** A *model element* is a record comprising: a universally unique identifier (UUID), a type (i.e. metaclass), and zero or more named 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 model of our running example (see Fig. 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.[3] Change actions applied to a design model yield a new version of it. Those change actions are formally defined as below.

Definition 3 (Change action)**.** A *change action* is one the following three types of primitive action: 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.

For example, Alice creates message *rinse* to component instance *ctrl*, which consists of the following sequence of primitive actions: adding a new message *rinse*, modifying its *receiveEvent* feature, and modifying the *represent* feature of Lifeline *ctrl*. For clarity, later in this paper we present changes in terms of their intuitive intended effects on the model, rather than these formal actions. For instance, deleting an inheritance relationship is described as a deletion, rather than as a modification to an attribute. We also view deleting a message as a single action.

Since a model is versioned, we are able to identify what exactly has been deleted in the previous version of the model. Similarly to the work of Blanc et al. (2008) we require that for a *delete(e,t)* action to be possible, the element being deleted must not be referred to elsewhere in the model (this condition is necessary in order to ensure that *add* is the opposite of *delete*). However, we note that, unlike Blanc *et al.*, we do not distinguish between features that are references and features that are values.

Definition 4 (Reverse action)**.** The *reverse action* of action a, denoted $\bar{a}$, is the action that has the opposite effect to a, as defined in the following table:

Action a	Reverse action $\bar{a}$
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)

The first step of our process (see Fig. 4) is to compute the differences between each revised model (e.g., versions 1 and 2, denoted M_1 and M_2) and the common ancestor (M) by leveraging state-of-the-art model differencing techniques (e.g. Abi-Antoun et al., 2006; Chen et al., 2004; Xing and Stroulia, 2005 or see Brosch et al., 2012b for a review of existing model differencing techniques). The difference is defined as below, where we use $M + \Delta$ to denote the model resulting from applying action sequence[4] Δ to model M.

Definition 5 (Difference between models)**.** 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. $M_{\text{old}} + \Delta = M_{\text{new}}$.

We extend the notion of the reverse of an action to also apply to sequences of actions.

Definition 6 (Reverse of action sequence)**.** The *reverse* of the action sequence Δ, denoted as $\overline{\Delta}$, is a sequence of actions that has the opposite effect to Δ, i.e. $(M + \Delta) + \overline{\Delta} = M$.

Note that $\overline{\Delta}$ is computed simply by reversing the sequence in Δ and replacing each action a with its reverse $\bar{a}$. 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. The two types of conflict are when one software engineer 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 engineers modify the same model

[3] In practice, most tools 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 a unique XMI identifier for each model element.

[4] We overload notation slightly: if A is an action *set* then we also use $M + A$ to denote the application of action *set* A to model M.

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 operation *process()* differently, which causes a conflict. Note that equivalent changes (e.g. creating a new class 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.

The next step of our approach (merging phase 1 in Fig. 4) involves computing the initial merged model by applying to the common ancestor model the union of all (non-conflicting) changes detected among the versions. Conflicting changes (modify–modify and delete–modify conflicts) are temporarily ignored at this stage: we do not employ any strategy nor ask the user to decide which changes shall be applied to the original model. User involvement, if desired, is possible later on when merging options are suggested to the user, which we will discuss in detail in Section 5. Therefore, this first phase of merging is fully automated. Before defining the initial merged model (Definition 7) we need to introduce notation to refer to conflicting and non-conflicting subsets of an action set. The intuition is that given two conflict-free action sets A_1 and A_2, some of the actions in A_1 conflict with A_2, whereas others do not, and we use $conf(A_1, A_2)$ to denote the set of actions in A_1 that conflict with A_2. We also define $nonconf(A_1, A_2)$ to be the set of actions in A_1 that do *not* conflict with A_2. Similarly, we have that $conf(A_2, A_1)$ (respectively $nonconf(A_2, A_1)$) is the set of actions in A_2 that conflict (respectively do not conflict) with A_1. We also extend this definition to be applicable to sequences of actions: $nonconf(\Delta_1, \Delta_2)$ is the set of all actions in the action sequence Δ_1 that do not conflict with any action in the sequence Δ_2.

Definition 7 (Initial merged model)**.** The *initial merged model* M_i of two variant models, M_1 and M_2, with respect to the common ancestor model M is defined as $M_i = M + (\Delta'_1 \cup \Delta'_2)$ where $\Delta'_1 = nonconf(\Delta_1, \Delta_2)$ and $\Delta'_2 = nonconf(\Delta_2, \Delta_1)$ are the actions applied to M to obtain M_i.

For example, the merged version (see Fig. 1) in the example presented in Section 2 is the outcome of the first phase. Note that our approach deals with *sets* of actions, rather than with sequences of actions. The effects of modifying field f of model element e_1, and then modifying field f' of entity e_2 are the same if the order if swapped. These actions are conflict-free. Although in general the order of actions matters, we only consider conflict-free action sets, and for these the order in which actions are applied makes no difference, so long as the creation of new entities precedes their use. In other words, we can consider, without a loss of generality, that the actions in a (conflict-free) set are always applied in the order: creation of entities first, then changes, and finally deletions. Conflict-free actions therefore can be automatically applied without user involvement. By contrast, conflicting actions may require human intervention (to choose which actions should be selected over the others) and thus are dealt with at the later stage of our merging process.

We need to treat conflicting and non-conflicting changes differently because non-conflicting changes have already been applied to the merged model (and, if they result in inconsistencies, then we *undo* them). On the other hand, conflicting changes have not been applied to the model, and where we can do so without creating inconsistencies, we want to apply a (non-conflicting) subset of those changes. Hence, both conflicting and non-conflicting changes give us a set of available actions that can be used for resolving inconsistencies in the initial merged model M_i.

Definition 8 (Available repair actionsΘ)**.** We define the set of available repair actions Θ for resolving inconsistencies in M_i to be $\Theta = \overline{\Delta'_1} \cup \overline{\Delta'_2} (\Delta_1 - \Delta'_1) \cup (\Delta_2 - \Delta'_2)$, where $\Delta'_1 = nonconf(\Delta_1, \Delta_2)$ and $\Delta'_2 = nonconf(\Delta_2, \Delta_1)$.

The set of available actions Θ contains the reverse of the non-conflicting changes between the model versions and the common ancestor, and the conflicting changes. Fig. 5 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 *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 Fig. 5).

The initial merged model together with conflicting changes stored in the set of available actions Θ can be viewed as a "wish list" containing all the things the users wanted. If there are no inconsistencies then the wish list is granted, i.e. we are done. However, if there are inconsistencies then variations of the initial model are computed (by including and/or excluding things from the wish list which is explored in detail below) to reflect consistent resolutions. The initial merged model is thus subjected to many combinations of changes. For each variation, a consistency check is performed (again using Egyed, 2006 which is very fast because the variations differ in few

Change set	Action	ID
$\overline{\Delta'_1}$ (The reverse of Alice's non-conflicting changes)	delete(Driver-inherit-Control, Generalization)	R1
	delete(rinsing, State)	R2
	delete(rinse, Transition)	R3
	modify(Transition[spin], source, rinsing, washing)	R4
	delete(rinse, Message)	R5
$\Delta_1 - \Delta'_1$ (Alice's conflicting changes)	*modify(Message[start], name, 'start', 'run'),*	R6
$\Delta_2 - \Delta'_2$ (Bob's conflicting changes)	*modify(Message[start], name, 'start', 'init')*	R7
$\overline{\Delta'_2}$ (The reverse of Bob's non-conflicting changes)	delete(Control-inherit-Driver, Generalization)	R8
	modify(Operation[init], name, 'init', 'run')	R9
	delete(stop, Message)	R10
	delete(turnOff, Message)	R11

Fig. 5. The set of available actions Θ for our running example.

changes only). In the next section, we will focus on the second phase of our merging approach (see Fig. 4) and explain how we employ a search technique to find the final candidate merged models.

5. State space search formulation

We formulate the second phase of merging in our framework as a state space search in which each state in the search space represents a (merged) model. The *initial state* represents the initial merged model M_i. In a *goal state*, the merged model is consistent.[5] In order to reach a goal state, the search may have to go through a number of *intermediate states* in which the model may contain some inconsistencies. Each state *S* is therefore associated with a list of *non-persistent* inconsistencies (e.g. $\mathtt{C1}(turnOff)$, $\mathtt{C1}(start)$, $\mathtt{C3}(Control)$, $\mathtt{C3}(Driver)$ in the initial state[6]) in the merged model (denoted as *S.inconsistencies*) and an inconsistency that we choose to fix (denoted as *S.inconsistency*), e.g. $\mathtt{C1}(turnOff)$. A state transition represents the *simulated* application of certain changes to the model in an attempt to repair the chosen inconsistency.

In our approach, the state space is implicit: we incrementally generate the next states as they are explored. An important part of the search is therefore determining the next states to be explored from a given state. Function *succ*() in Algorithm 1 describes how such successor states are generated in our approach. This function takes as input a given state *S*, the set of available repair actions Θ previously obtained in the initial merging process, and the path *P* (represented as a list of location–action pairs) from the initial state to state *S*. The function returns as output a list of state-actions pairs $\langle S', A\rangle$ where $S' = S + A$ (denoting that state S' is generated by applying actions *A* to the model at state *S*), which is ordered (line 10 in Algorithm 1) by a preference ordering ($\preceq$). The preference ordering (see Definition 9 below), which is a partial order, reflects a difference between conflicting and non-conflicting actions. Non-conflicting actions in Θ represent *undoing* changes to the model, and therefore we prefer to do as few of these as possible. On the other hand, conflicting actions in Θ are the *application* of (some of) the changes that the modelers have made, and therefore we want to apply as many of these as possible. In other words, a solution is more preferred if it has *fewer* non-conflicting actions, and *more* conflicting actions (without actually having a conflict). We define the preference ordering below.

Algorithm 1: succ(*S*, Θ, *P*) and getCAs(Θ, *P*)

```
function succ(S, Θ, P): generate the successor states
                        from a given state
1.  availActions, succs:= ∅, ∅
2.  if S.inconsistencies ≠ ∅ then
3.      S.inconsistency := S.inconsistencies.removeFirst
          Inconsistency()
4.      iScope := scope(S.inconsistency)
5.      for each l ∈ iScope do
6.          availActions := availActions ∪
              {(locn(a), a)|a ∈ Θ ∧ locn(a) = l ∧ (l, a) ∉ P}
7.  for each A ⊆ availActions do
8.      if A is conflict-free then
9.          succs := succs ∪ {⟨S', A⟩} where S' = S + A
10. sort succs by ⪯ (which is a partial order)
11. return succs

function getCAs(Θ, P): generate options for resolving
                       conflicting actions
1.  availActions, CA := ∅, ∅
2.  for each conflicting pair of actions ⟨a_1, a_2⟩ ∈ Θ do
3.      if (locn(a_1), a_1) ∉ P ∧ (locn(a_2), a_2) ∉ P then
        /* Have neither a_1 nor a_2: add them */
4.          availActions := availActions ∪ {(locn(a_1), a_1),
              (locn(a_2), a_2)}
5.  for each A ⊆ availActions do
6.      if A ≠ ∅ and A is conflict-free then
7.          CA := CA ∪ {A}
8.  sort CA by ⪯ (which is a partial order)
9.  return CA
```

Definition 9 (Preference ordering $\preceq$)**.** We define $A_1 \preceq A_2$ ("A_1 is preferred to A_2", where A_1 and A_2 are each a conflict-free set of actions) as holding iff $\mathit{nonconf}(A_1, A_2) \subseteq \mathit{nonconf}(A_2, A_1) \wedge \mathit{conf}(A_2, A_1) \subseteq \mathit{conf}(A_1, A_2)$.

Observe that $A_1 \preceq A_2$ does *not* imply that $A_1 \subseteq A_2$, and we therefore define a stronger version of preference that does imply that A_1 is a subset of A_2. Also, note that $A_1 \preceq A_2$ is a partial order.

Definition 10 (Strong preference ordering $\sqsubseteq$)**.** We define $A_1 \sqsubseteq A_2$ (where A_1 and A_2 are each a conflict-free set of actions) to hold iff $nonconf(A_1, A_2) \subseteq nonconf(A_2, A_1) \ \wedge \ conf(A_2, A_1) = conf(A_1, A_2)$.

Observe that $A_1 \sqsubseteq A_2$ implies that $A_1 \subseteq A_2$ (but the converse does not hold since the difference could involve conflicting actions).

The *succ*() function basically builds up a set of available actions that can be applied to the merged model associated with a given state *S* (lines 2–6), and for each combination of those actions that is conflict-free (i.e. does not include actions that conflict with each other), it creates a new successor state (lines 7–8). Two actions conflict with each other in the following situations: (i) one action modifies a feature of a model element deleted by the other action; and (ii) when both actions modify the same model element feature in different ways. It first checks if the model at state *S* contains some (non-persistent) inconsistencies, one of which (*S.inconsistency*) is chosen and we need to identify the available actions that can potentially fix it. As previously discussed, an inconsistency can be fixed only by changing one or more model elements that the inconsistent constraint accessed during its evaluation, which is the scope of the constraint. The scope of a constraint instance is automatically computed by observing which model elements are accessed during the evaluation of the constraint. For instance, the evaluation of constraint C1, i.e. *self*.receiveEvent.covered → *forAll*(represents.type.ownedOperation → *exists*(name = self.name)), on message *start* accesses this message first. It then iterates over all Lifelines that the message is sent to (due to the universal quantifier *forAll* – UML allows a message to be sent on more than one lifeline). The lifelines are accessed through the properties *receiveEvent* (referencing to another model element of type *MessageOccurrenceSpecification*) and *covered* of that *MessageOccurrenceSpecification* from the message *start*. The evaluation then iterates (due to the existential quantifier *exists*) over the operations of the class that is the type of the lifeline. This is done by accessing the properties *represents* (instance *gui* of the lifeline), type (class *GUI*), and *ownedOperation* (operation *init*()). The scope of constraint $\mathtt{C1}(start)$ is therefore the model elements {*start, gui, GUI, init*()}.

We obtain the scope elements of *S.inconsistency* (line 4) and retrieve actions in the available action set Θ that affect any of those scope elements (lines 5–6). In order to avoid the search from falling into a cycle, we discard any actions that have been previously applied in the path leading to this state. For example, suppose that we choose to explore options for fixing $\mathtt{C3}(Control)$ in the model where $\mathtt{C1}(turnOff)$ has been fixed by deleting the *turnOff* message.

[5] Note that persistent inconsistencies are ignored since the search only looks for actions in the set of available set actions Θ. These actions are not able to resolve persistent inconsistencies.

[6] $\mathtt{C1}(wash)$ is persistent and hence ignored.

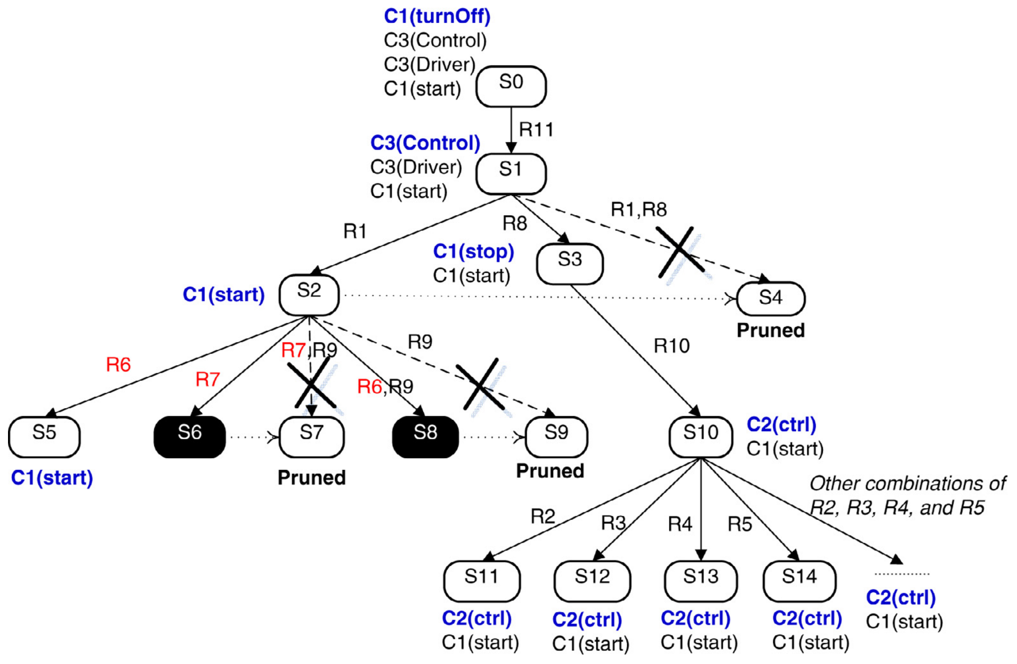


Fig. 6. An example of the search tree (goal states are indicated using white on black).

Since the scope of constraint C3(*Control*) is *Driver.generalization*[7] and *Control.generalization*, cross-checking the available action set Θ (see Fig. 6) gives us two available actions: deleting the generalization from *Driver.generalization* (R1), and deleting it from *Control.generalization* (R8).

The final part of the function (lines 7–9) uses the available actions to generate all possible combinations of non-conflicting actions that can potentially resolve the *S.inconsistency*. Each of those combinations gives a new successor state and the set of actions that are applied to the model to bring it to this state. For example, given the available actions $R1$ and $R8$, there are three possible combinations: applying $R1$, applying $R8$, or applying both $R1$ and $R8$, which generates three successor states ($S2$, $S3$, and $S4$) of $S1$ (see Fig. 6).

A solution path (from the initial state to the goal state) represents one option for resolving inconsistencies and conflicts in the merging. There can be multiple goal states (and multiple solution paths), representing multiple valid ways of resolving inconsistencies and conflicts. In the next section, we will describe how we explore the state space in searching for all the possible goal states.

5.1. Incremental search space exploration

The search for a consistent version of the merged model follows a depth-first-search style (see procedure *search*() in Algorithm 2). The search progresses by expanding the first successor of a given state S and going deeper and deeper until any of these scenarios are encountered: (a) a goal state is found and all conflicting-changes have been tried; (b) it moves to a state but cannot resolve the inconsistency that was chosen to be fixed; (c) it reaches a state that has no successors. Then the search continues (noting, in case (a), that a goal state was found), returning to the most recent node it has not finished exploring. The search takes into account both negative and positive side effects of an inconsistency resolution. We now describe this process in detail.

The search starts by generating a list of successors that are not explored from the current state S (lines 1–2 of procedure *search*()). We then move to explore each of those successors (line 3). Lines 4, 16–18, and 20–21 relate to pruning and are explained later. The search process marks the current state S' as having been explored (line 5), and then requests the consistency checker to evaluate the updated model (line 6). The evaluation is done *incrementally* since only constraint instances whose scope elements are affected by

Algorithm 2: search(S, Θ, P)

```
procedure search($S, \Theta, P$) navigate the search space
          from a given state
1.  successors := succ($S, \Theta, P$)
2.  succs := $\{\langle s, a\rangle \mid \langle s, a\rangle \in$ successors $\wedge \neg s.explored\}$
3.  for each $\langle S', A\rangle \in$ succs do
4.     if $S'.pruned$ then continue /* skip pruned */
5.     $S'.explored := true$
6.     $S'.inconsistencies := S'.updateConstraints()$
7.     if $S'.inconsistencies = \emptyset$ then
8.        $S'.isGoal := true$
9.        $CA := getCAs(\Theta, P)$
10.       for each $ca \in CA$ do
11.          apply $ca$ to $S'$ to get $S''$
12.          $S''.inconsistencies := S''.updateConstraints()$
13.          if $S''.inconsistencies = \emptyset$ then
14.             $S''.isGoal := true$
15.             $S'.isDominated := true$
16.             $CA := \{ca' | ca' \in CA \wedge \neg ca \preceq ca'\}$
17.             for each $\langle S''', A'''\rangle \in$ succs do
18.                if $A \preceq A'''$ then $S'''.pruned := true$
19.    else if $S.inconsistency \notin S'.inconsistencies$
          then /* successfully fixed constraint: prune,
             then fix remaining constraints */
20.       for each $\langle S''', A'''\rangle \in$ succs do
21.          if $A \sqsubseteq A'''$ then $S'''.pruned := true$
22.       search($S', \Theta, P \cup A$)
23.    else /* do nothing: continue to next iteration */
```

Note initial call is search(S_0, Θ, ∅) where S_0 is the initial state corresponding to the initial merged model and Θ is the set of available actions.

[7] *A.generalization* denotes the set of generalization/super classes of *A*.

the applied actions are re-validated. In the new state, the chosen inconsistency may be absent (i.e. the fix is successful) or still present (the fix is not successful), and some other inconsistencies may also be fixed (positive effects) or some new inconsistencies may have been introduced (negative effects). If there is no inconsistency found in the updated model (i.e. the fix is successful and does not cause any negative side-effects, line 7), then we will move to explore whether there are any conflicting actions that have not yet been tried, and which can be applied without introducing inconsistencies. This is done by calling function *getCAs*() (line 9) to get possible actions sets, and then for each $ca \in getCAs()$ we apply *ca* (lines 10–11) and, if the resulting model is consistent, we mark it as a goal state (lines 12–14). Since applying additional conflicting actions results in a more preferred solution, we also mark S' as being a dominated solution (line 15).

If the updated model is inconsistent, there are two possibilities that should be considered. If the model was not consistent and the inconsistency that we attempted to fix (by moving to this successor state) still exists, meaning that the fix was not successful, we will continue to explore other successor states (line 23). Otherwise, i.e. if the fix successfully resolves the selected inconsistency, then the search continues exploring to fix one of the remaining inconsistencies (line 22).

We also include a simple pruning mechanism in the algorithm based on the idea that whenever the violation of a constraint *C* is successfully repaired by applying an action set *A*, then we do not explore any successors generated by an action set A' that is less preferred than *A*. The rationale is the principle of minimality: that if we can successfully repair an inconsistency by applying action set *A*, then we should not consider applying a less preferred action set (i.e. one that has unnecessary additional non-conflicting actions). The pruning is implemented by lines 18 and 21 of *search*(). In line 21, when we have repaired an inconsistency, and are about to continue exploring the repair of other inconsistencies, we go through the *succs* set and mark the states generated by A' (where $A \sqsubseteq A'$) as being pruned. Note that the successor states are pre-sorted in order of preference. In line 18, when we have found a goal state, we also check for other solutions that are less preferred, and prune them. Note that here, since we have found a goal-state, we can use a slightly stronger notion of preference that regards a state as being preferred not just if it has fewer non-conflicting actions, but also if has additional conflicting actions ($\preceq$). We also add a check for pruned states (line 4 of *search*()). Finally, we also perform pruning on the set *CA* (line 16) by removing possible sets of actions that are less preferred than the current (successful) conflicting action set *ca*.

The function *getCAs*() returns possible subsets of conflicting actions that are conflict-free, i.e. do not contain two actions that conflict with each other. For each pair of conflicting action, for example *R*6 and *R*7, if neither action has already been performed, then both actions are added to the set of available actions. Once the set of available actions has been constructed, the function returns all action sets $A \subseteq availActions$ which are conflict-free. The search process converges because the set of available actions (availActions), which is used in Algorithm 1 (succ), decreases monotonically.

Fig. 6 shows the search tree of our running example. The search starts at *S*0 (the initial merged model) and chooses to fix inconsistency $\mathtt{C1}(turnOff)$ (among the four inconsistencies[8] $\mathtt{C1}(turnOff)$, $\mathtt{C3}(Control)$, $\mathtt{C3}(Driver)$, and $\mathtt{C1}(start)$). There is only one successor state *S*1 (corresponding to applying *R*11) in which inconsistency $\mathtt{C1}(turnOff)$ is resolved. At state *S*1, the search continues and chooses to fix inconsistency $\mathtt{C3}(Control)$ (among the remaining three inconsistencies at this state). As discussed earlier, there are three successor states, and the search explores *S*2 first (i.e. performing *R*1). At state *S*2, the merged model no longer has the circular inheritance issue (*Driver* is no longer a subtype of *Control*) and there is only one remaining inconsistency i.e. $\mathtt{C1}(start)$ (message *start* does not match any operation in class *GUI*). Since the selected *S.inconsistency* was repaired using *R*1, we prune {*R*1, *R*8} (dashed arrow from *S*2 to *S*4 in Fig. 6). The search then generates successor states (*S*5, *S*6, *S*7, *S*8, and *S*9) for *S*2 from all possible combinations of the available actions *R*6, *R*7 and *R*9. Note that since *R*6 and *R*7 are conflicting changes due to modifying the same location (i.e. *start.name*) differently, they cannot be part of the same combination (i.e. the combinations {*R*6, *R*7} and {*R*6, *R*7, *R*9} are excluded). The search then moves to explore *S*5 first (i.e. performing *R*6). At state *S*5, message *start* is now renamed to *run*, which still does not match the name of the message (i.e. *init*). This fix is not successful, and thus the search backtracks to state *S*2 (unapplying *R*6) and moves to explore *S*6 (applying *R*7). At state *S*6, message *start* is now renamed to *init*, which matches the message's name and consequently resolves the inconsistency $\mathtt{C1}(start)$. Since there is no inconsistency at *S*6, the state is marked as a goal state.[9] At this point we know that *R*7 suffices to repair the inconsistency, and so applying both *R*7 and *R*9 to resolve the constraint is unnecessary. We therefore prune state *S*7 (indicated by the dashed arrow from S5). The search then moves to explore the remaining successor states and discovers that state *S*8 (both the operation and message are renamed to *run*) is also a goal state. Since *R*6 is a conflicting change (with *R*7), applying *R*6 and *R*9, which results in goal state *S*8, is preferred to applying only *R*9, and thus state *S*9 is pruned.

After finishing exploring all successor states of *S*2, the search moves to *S*3, which causes a new inconsistency $\mathtt{C1}(stop)$ (since class *Control* no longer inherits operation *stop*() from class *Driver*). The search then chooses to fix $\mathtt{C1}(stop)$ by applying *R*10 to the model (deleting message *stop* to move to state *S*10), which causes a new inconsistency $\mathtt{C2}(ctrl)$ since the sequence of incoming message to component *ctrl* no longer matches the transitions in the state machine diagram). The search continues generating and exploring successors of state *S*10 to fix inconsistency $\mathtt{C2}(ctrl)$ by trying all different components of the available actions *R*2, *R*3, *R*4, and *R*5. However, none of these components resolves the consistency.

Returning now to the example scenario, Alice and Bob, having run our merging tool, would be presented with two alternative merging options (see Fig. 7), each of which corresponds to a goal state found by our algorithm. The left side of Fig. 7 corresponds to goal state *S*6: the changes *R*11 (deleting message *turnOff*), *R*1 (delete the inheritance relationship between *Driver* and *Control*) and *R*7 (renaming message *start* to *init*) have been applied to the initial merged model. The right hand side of the figure corresponds to goal state *S*8, where instead of *R*7, changes *R*6 (renaming message *start* to *run*) and *R*9 (rename method *init*() to *run*()) have been applied.

Our merging algorithm returns all possible minimal goal states, which requires an extensive exploration of the search space. Since any of the alternative consistent versions found by our approach may be chosen by the user, a "safe" solution is presenting all of them, which might be a large number, to the user and letting them make the choice. While future work would involve investigating how to present such solutions in an effective and user-friendly manner, a heuristic can reduce the solution space and can also improve the performance of the search. The heuristic reflects the fact that one would want to integrate in the merge as much new information from the versions as possible. Applying this heuristic to our search implies that we now look for *optimal* solution(s), which contain the minimal number of non-conflicting changes that need to be unapplied and the maximal number of conflicting changes that can be applied. The metric is then represented as a *distance* of the path from the initial state to a goal

[8] Recall that $\mathtt{C1}(wash)$ is detected as being "persistent" and hence excluded from consideration.

[9] Note that in this example, there are no unapplied conflicting actions, since *R*7 has already been applied, and therefore *R*6 cannot be also applied.

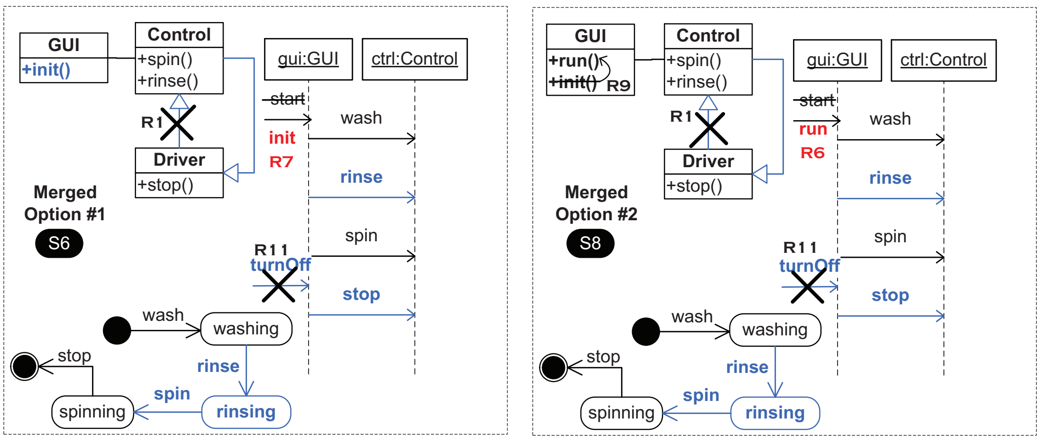


Fig. 7. Options presented to Alice and Bob.

state, and the optimal solution(s) would be the one(s) with the shortest distance. Note that applying a conflicting action should be seen as a *good* thing (i.e. negative cost) since we want to apply as many of those actions as possible. We may therefore define the notion of distance to assign a cost (e.g. 10 units) to unapplying a change, and assign a cost (e.g. 1 unit) to each conflicting action that is not applied. These numbers do not correspond to any real cost, and are simply used to compare between unapplying non-conflicting changes and applying conflicting changes. It is worth noting that the selection of the specific cost ratio is somewhat arbitrary, and thus we could allow the user to specify it. Using this heuristic, we could identify and present the best options to the software engineers. Alternatively, the software engineers could also be given all the options ranked based on the same distance measure used by our heuristic.

5.2. Repair generation

As presented in the previous section, the exploration to find out how non-persistent inconsistencies in the initial merged model M_i can be resolved using available actions in Θ are done in an incremental manner. 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). This approach utilizes the scope of a constraint to reduce the number of combinations that need to be enumerated to identify fixes for the constraint violation: from $2^{\#changes}$ 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 (if all actions in Θ access a constraint's scope).

In practice, there is only a very small number of combinations of available actions which can resolve an inconsistency. In fact, previous work (Reder and Egyed, 2012) has shown that independent of the model size, there are on average only 12 possible repairs per inconsistency[10] (see Fig. 8). Thus, it is inefficient to enumerate through a very large number of combinations, only to find 12 of which can actually resolve an 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

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

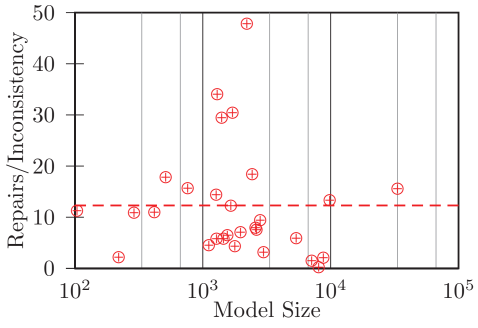


Fig. 8. Average number of repairs per inconsistency (dashed line = average across all models).

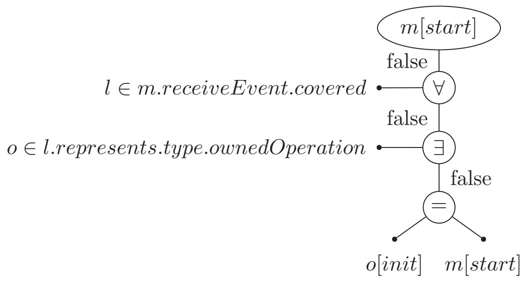
exactly which parts of the inconsistency must be repaired. In the following we briefly outline the approach presented in Reder and Egyed (2012) and explain how this approach has been extended to be able to make repair actions concrete (i.e. a concrete value is known).

The basis for the repair generation is the so called validation tree (Reder and Egyed, 2012), which is created when a constraint instance is first evaluated. For illustration we formalize constraint C1 as below.

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

This constraint is written for a context element of the type *Message*. The condition of this constraint is a Boolean expression that is validated on the model. The constraint reveals the sequence in which model elements are accessed and the specific properties are used. For example, this constraint is a universal quantifier (∀) iterating over a set of lifelines *l*. These lifelines are found by first validating the "receiveEvent" property of message "m", which references another model element of type "MessageOccurenceSpecification"; and then by validating the "covered" property of that "MessageOccurenceSpecification", which references a model element of type "Lifeline". Details on the model elements and their properties can be found in Object Management Group (2004). The universal quantifier then iterates over all lifelines found to ensure that all lifelines satisfy the quantifier condition. This condition is another existential quantifier ($\exists o \in l.represents.type.ownedOperation$).

The validation tree which was created when evaluating constraint C1(*start*) is shown in Fig. 9.



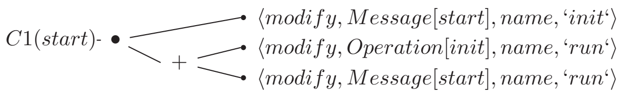
m[start]
false
l ∈ m.receiveEvent.covered
∀
false
o ∈ l.represents.type.ownedOperation
∃
false
=
o[init]
m[start]

Fig. 9. Validation tree for C1(*start*). The circular nodes are Boolean expressions and the edges list the validation results.

The validation starts at the model element *start* (the root node of the validation tree). The first operation executed is the universal quantifier (∀) 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 (∃) 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 Reder and Egyed (2012).

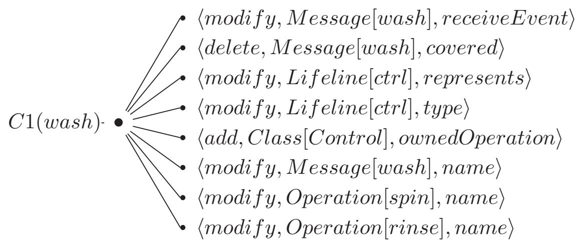
A repair tree is built based on the validation tree. The nodes of the repair tree are directly derived from the validation tree: ∀ and ∧ are mapped to + (denoting combinations of repair actions), while ∃ and ∨ are mapped to • (denoting alternative repair actions). Each branch of a validation tree has an expected and validated result. Note that in Fig. 9 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 inversion of the expected result (*true*↔*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: ∀ → *delete*, ∃ → *create*, and = → *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 Reder and Egyed (2012) are *abstract* repairs. In this example, one abstract repair generated (as in Reder and Egyed, 2012) 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 Reder and Egyed (2012) 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 message *start* to *init* (R7) is an action in Θ (see Fig. 5), which is an instance of the abstract repair *modify(Operation[init], name)*. Another abstract repair suggested by the repair tree is the renaming of both message *start* and operation *init()* to the same name. Based on the available actions in Θ, another concrete repair for C1(*start*) is therefore *modify(Operation[init], name, 'init', 'run')* and (+) *modify(Message[start], name, 'start', 'run')*. Fig. 10 shows the full concrete repair tree for the validation shown in Fig. 9 using the available actions in Θ. This repair tree represents two alternative, available repair plans for resolving inconsistency C1(*start*): renaming message *start* to *'init'* (*R*7), or renaming both the message and operation *init* to *'run'* (*R*6 and *R*9).



C1(start)-
+
⟨modify, Message[start], name, 'init'⟩
⟨modify, Operation[init], name, 'run'⟩
⟨modify, Message[start], name, 'run'⟩

Fig. 10. Concrete repair tree for C1(*start*).



C1(wash)-
⟨modify, Message[wash], receiveEvent⟩
⟨delete, Message[wash], covered⟩
⟨modify, Lifeline[ctrl], represents⟩
⟨modify, Lifeline[ctrl], type⟩
⟨add, Class[Control], ownedOperation⟩
⟨modify, Message[wash], name⟩
⟨modify, Operation[spin], name⟩
⟨modify, Operation[rinse], name⟩

Fig. 11. Abstract repair tree for C1(*wash*).

The repair generator is used in generating the successor states from a given state in our merging exploration algorithm, i.e. function *succ*() in Algorithm 1. More specifically, lines 4–9 of function *succ*() are replaced with a call to the repair generator to obtain the exact combinations of actions (from the set of available actions Θ) which can fix inconsistency *S.inconsistency*. For example, if we rely on the repair generator instead of the constraint scope, there would be only two successor states of state *S*2 in Fig. 6: state *S*6 corresponding to repair *R*7 and *S*8 corresponding to repair *R*6 ad *R*9.

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 (where 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 C1(*wash*) is shown in Fig. 11, which suggests a number of alternative ways to resolve this inconsistency. For example, repair action ⟨*modify, Message*[*wash*], *name*⟩ suggests that message "wash" needs renaming. Other alternatives to resolve this inconsistency such as renaming operation "spin" (⟨*modify, Operation*[*spin*], *name*⟩) or operation "rinse" (⟨*modify, Operation*[*rinse*], *name*⟩) of class "Control" are also suggested. Again note that the repair action is abstract since it does not reveal what string to rename it to. Existing approaches for computing concrete repairs may be used to complement our repair trees with concrete values and this will be the focus of our future work.

6. Evaluation

We have developed a prototype implementation of our approach and integrated it with IBM Rational Software Architect (RSA) (IBM, 2013) and use the existing merging functionality provided with IBM RSA for obtaining the initial merged model. IBM RSA uses the three-way merging approach which is compatible with our approach. It also uses the Model/Analyzer tool for checking inconsistencies (Reder and Egyed, 2010) (an implementation of Egyed, 2006). The prototype currently focuses on implementing the merging algorithms (i.e. Algorithms 1 and 2 in the paper) and is available at http://www.uow.edu.au/~hoa/modelmerger/. We now present the proof for the correctness of our algorithm and discuss its scalability in practice.

6.1. Correctness

We want to show that Algorithm 1 is *sound*, i.e. it only proposes goal states that are consistent and are derived from actions in Θ; and that it is *complete*, i.e. it finds all (preferred) goal states. Note that we do not want the algorithm to be complete in the sense of finding all goal states, but in the sense of finding all *preferred* goal states, which allows pruning to be done.

Theorem 1 (Soundness)**.** The *search()* algorithm is *sound*: it only flags as goal states those states that are derived by applying a subset of the actions in Θ and where all non-persistent inconsistencies have been resolved.

Proof. The algorithm only marks a state as a goal state if it has no remaining inconsistencies (checked in lines 7–8 and 13–14 of Algorithm 2). Furthermore, the actions that the algorithm considers are derived from those in Θ (in the *succ* and *getCAs* functions), and so it only ever applies actions from Θ. □

Whereas showing soundness is straightforward, proving completeness is less straightforward. We show the completeness of the algorithm in two stages. Firstly, we define a variant algorithm that does not do any pruning, and that does not apply conflicting actions. This simpler algorithm (denoted below *search′*) consists of the **bold** line numbers in Algorithm 2. We then proceed to prove that this simpler algorithm is complete (as defined below). In the second stage, we argue that the pruning mechanism and the application of additional conflicting actions only ever leave out a solution when a better solution exists, and that the algorithm with pruning and conflicting actions is therefore also complete (in a slightly different sense, defined below).

We begin by defining when an action set $A \subseteq \Theta$ is a *solution*, and when it is a *non-redundant solution*. We use "solution state" to refer to the state that results from applying a solution (i.e. if A is a solution for some constraints with respect to starting state S_0, then $S = S_0 + A$ is a solution state). We use the notation $S \vDash C$ to indicate that constraint(s) C hold in state S.

Definition 11 (Solution)**.** An action set $A \subseteq \Theta$ (where A must be conflict free) is a *solution* for $C_1 \ldots C_n$ with respect to starting state S iff all the C_i are satisfied in $S + A$.

The intuition behind non-redundant solutions is that the algorithm is not complete in the sense of finding all solutions, since a solution can be extended with additional, irrelevant, actions. Rather, the algorithm is complete in the sense of finding all non-redundant solutions.

Definition 12 (Non-redundant solution)**.** An action set $A \subseteq \Theta$ is a **non-redundant solution** for $C_1 \ldots C_n$ w.r.t state S iff: A is a solution for $C_1 \ldots C_n$ with respect to S; and there does not exist $A' \subset A$ which is a solution for $C_1 \ldots C_n$, i.e. A is minimal.

Before we proceed to tackle completeness, we first establish two useful lemmas. The first shows that we can decompose solutions. The second establishes that the *succ(S, Θ, P)* function finds all solutions for a single constraint (as long as P does not prevent relevant actions from being selected).

Lemma 1 (Decomposition of non-redundant solutions)**.** Any set of actions A that is a non-redundant solution for $C_1 \ldots C_n$ with respect to S_0 can be decomposed into three parts, A_1, A_{2-n}, A^+:

$$\begin{tikzcd}
S_0 \arrow[r, "A"] \arrow[d, "A_1"'] & S' \vDash C_1 \ldots C_n \\
S_1 \vDash C_1 \arrow[r, "A_{2-n}"'] & S_2 \vDash C_2 \ldots C_n \arrow[u, "A^+"']
\end{tikzcd}$$

such that A_1 is a non-redundant solution for C_1 with respect to S_0; A_{2-n} is a non-redundant solution for $C_2 \ldots C_n$ with respect to $S_1 = S_0 + A_1$; and A^+ is a finite sequence of zero or more sets $\langle A_1^+, \ldots, A_k^+ \rangle$, where, if $k \geq 1$, A_1^+ is a non-redundant solution for C_1 with respect to $S_2 = S_1 + A_{2-n}$, and each A_i^+ $(i > 1)$ is a non-redundant solution for one of the constraints C_j $(1 \leq j \leq n)$ with respect to S_{i-1}^+ (where $S_1^+ = S_2 + A_1^+$ and $S_i^+ = S_{i-1}^+ + A_i^+$ for $i > 1$).

Proof. We construct A_1, A_{2-n} and A^+ from A as follows. We know that applying A to S_0 repairs C_1. However, A may contain actions that are unnecessary for repairing C_1 (since A also repairs $C_2 \ldots C_n$). We therefore construct A_1 by simply removing those unnecessary actions, resulting in a non-redundant A_1. Similarly, we know that, since A is a solution for $C_1 \ldots C_n$ w.r.t. S_0, then $A - A_1$ is a solution for $C_2 \ldots C_n$ w.r.t. $S_1 = S_0 + A_1$. However, $A - A_1$ may contain actions that are not necessary for repairing $C_2 \ldots C_n$. We therefore remove these actions, giving a non-redundant A_{2-n}. Finally, we consider A^+. If there are any actions remaining (i.e. $A^+ = A - (A_1 \cup A_2)$ is non-empty), then we must have that C_1 is violated in S_2. We know that A^+ is a solution for C_1 w.r.t. S_2 (since A is a solution for $C_1 \ldots C_n$ w.r.t. S_0), but A^+ may contain non-essential actions for repairing C_1. We obtain a non-redundant solution for C_1 w.r.t. S_2 by selecting only the essential actions from A^+ into A_1^+ and leaving the remaining actions in A^+. We then consider $S_1^+ = S_2 + A_1^+$ and the remaining actions A^+. As before, if A^+ is non-empty then there must be a violated constraint C_j, and so we form A_2^+ by selecting those actions necessary for repairing C_j w.r.t. S_i^+. This process is applied until A^+ is empty. Termination is guaranteed since A is finite, and since each A_i^+ is non-empty, so eventually all actions in A^+ are allocated to an A_i^+. □

Lemma 2 (Correctness of succ)**.** If $A \subseteq \Theta$ is a non-redundant solution for the single constraint C_1 with respect to starting state S, then $\langle S + A, A \rangle \in succ(S, \Theta, P)$ for any P that satisfies $P \cap A = \emptyset$.

Proof. The function *succ*() generates all possible non-conflicting subsets of availActions$\subseteq\Theta$. Since *succ* generates all subsets of availActions, it will therefore generate $\langle S + A, A \rangle$ as long as A does not involve any actions that are in Θ but not in availActions. To show that A only requires actions that are in availActions we consider how *succ* defines availActions: it computes it by filtering out actions in P, which we know does not intersect with A ($P \cap A = \emptyset$), and it limits to actions that are in the scope of C_1. Actions that are outside C_1's scope cannot affect its truth, and since A is non-redundant, it cannot contain such actions: if it did, then they could be removed without affecting C_1, and hence the shorter sequence would still be a solution, and A would not be non-redundant. Therefore, limiting to those actions in availActions does not reduce the options to repair C_1, and hence *succ* is complete as desired. □

We now show completeness of the modified algorithm, *search′*, which omits pruning and adding conflicting actions. We want to show that *search′* is complete in the sense that when it is invoked as *search′*(S_0, Θ, ∅), then all states S' resulting from applying a non-redundant solution $A \subseteq \Theta$ to the initial state S_0 (i.e. $S' = S_0 + A$) will be flagged as goal states by the algorithm. In fact, we actually prove a slightly more general result: that the call to *search′*(S_0, Θ, P) flags S' as a goal state, as long as $P \cap A = \emptyset$. The desired result is clearly a corollary of this stronger result (just let $P = \emptyset$, which trivially satisfies $\emptyset \cap A = \emptyset$).

Theorem 2 (Completeness of search′)**.** Algorithm *search′* is *complete*: whenever $A \subseteq \Theta$ is a non-redundant solution for $C_1 \ldots C_n$ with respect to state S_0, then, for any P satisfying $P \cap A = \emptyset$, *search′*(S_0, Θ, P) flags the state $S' = S_0 + A$ as a goal state.

Proof. By induction over the number of constraints.

Base case: one constraint C_1. In this case we need to show that if A_1 is a non-redundant solution for C_1 with respect to state S_0, then *search′*(S_0, Θ, P) flags $S_1 = S_0 + A_1$ as a goal state (as long as $P \cap A = \emptyset$). We do this by simply applying Lemma 2 and observing that *search′* operates by calling function *succ*(), and then applying each option, so since $\langle S_0 + A_1, A_1 \rangle \in succ(S_0, \Theta, P)$ the algorithm will eventually apply A_1. Since A_1 is a solution, the constraint C_1 holds in S_1, and therefore the condition in line 7 of *search*′ holds, and S_1 is flagged as a goal state in line 8, as desired.

Induction hypothesis: Assume that whenever A' is a non-redundant solution for $C_2 \ldots C_n$ w.r.t. state S_1, then $search'(S_1, \Theta, P)$ (for $P \cap A' = \emptyset$) reaches state $S' = S_1 + A'$ (note that this is a weaker condition than flagging S' as a solution).

Induction proof: We need to show that if A is a non-redundant solution for $C_1 \ldots C_n$ with respect to state S_0, then $search'(S_0, \Theta, P)$ flags the state $S' = S_0 + A$ as a goal state (as long as $P \cap A = \emptyset$). We reason by following the operation of *search*′:

1. The algorithm first computes the set of successors using $succ(S_0, \Theta, \emptyset)$, focusing on constraint C_1. From Lemma 1 we know that we can decompose A into three disjoint parts: A_1 (a non-redundant solution for C_1 w.r.t S_0), A_{2-n} (a non-redundant solution for $C_2 \ldots C_n$ w.r.t $S_1 = S_0 + A_1$), and a (possibly empty) sequence of sets A^+. From Lemma 2 we know that $\langle S_0 + A_1, A_1 \rangle \in succ(S_0, \Theta, \emptyset)$.
2. The algorithm iterates through the succs set, eventually processing A_1
3. When dealing with A_1 it checks whether there are any violated constraints in $S_1 = S_0 + A_1$. There are two possibilities here.
   (a) The first possibility, which is unlikely, is that applying A_1 fixes all of the constraints $C_1 \ldots C_n$. In this case the first if statement's condition holds (at line 7), and the state S_1 is flagged as a goal state as desired. This case corresponds to an empty A_{2-n} and A^+, i.e. $A = A_1$.
   (b) The second possibility is that applying A_1 fixes C_1, but not $C_2 \ldots C_n$. In this case the condition of the "else if" holds (line 19), and the algorithm proceeds with the recursive call $search'(S_1, \Theta, A_1)$.
4. Assuming case (b), the induction hypothesis is now applied, and therefore we know that the state $S_2 = S_1 + A_{2-n}$ is reached (note that $A_1 \cap A_{2-n} = \emptyset$). There are now two cases.
   (i) Constraint C_1 holds in state $S_2 = S_1 + A_{2-n}$. In this case the if statement at line 7 is true, and S_2 is flagged as a goal state as desired (line 7). This case corresponds to an empty A^+ (i.e. $A^+ = \langle\rangle$).
   (ii) Constraint C_1 does not hold in state S_2. In this case the algorithm proceeds to call $search'(S_2, \Theta, A_1 \cup A_{2-n})$. This recursive call uses $succ()$ to find A_1^+ (since A_1^+ is a non-redundant solution for C_1 w.r.t. S_2 Lemma 2 applies), then applies A_1^+ to fix C_1, resulting in a state S_2^+. If we have cumulatively applied A to S_0 (i.e. there is no A_2^+, so $A = A_1 \cup A_{2-n} \cup A_1^+$) then, since A is a non-redundant solution for $C_1 \ldots C_n$ w.r.t. S_0, we know that $S_2^+ \vDash C_1 \ldots C_n$, so the condition in line 7 holds, and the algorithm flags S_2^+ as a goal state as desired (line 8). Otherwise the algorithm proceeds to apply each A_i^+ in turn (using Lemma 2, since each is a non-redundant solution), and eventually reaches the state where A has been cumulatively applied, all constraints are repaired, and this state is flagged as a goal state, as desired. □

We have shown that the simplified algorithm *search*′, that does not do pruning and does not apply additional conflicting actions (using getCAs), is complete in the sense of finding all *non-redundant* solutions. We now extend this, by arguing that adding pruning and conflicting actions does not cause the algorithm to miss any *preferred* solutions. Note that this involves a slight change in the notion of completeness: the full algorithm does not find all non-redundant solutions, since it is possible for pruning to cause a non-redundant solution to be missed. However, this will only happen when the solution being missed is less preferred than another solution that is found, and so we show that the full *search* algorithm is complete in the sense of finding all *preferred* solutions.

Theorem 3 (Completeness of search)**.** The full algorithm is *complete*: whenever $A \subseteq \Theta$ is a non-redundant solution for $C_1 \ldots C_n$ with respect to state S_0, then either $S' = S_0 + A$ is flagged as a goal state, or there exists another state $S'' = S_0 + A''$ that is the result of applying to S_0 an action set A'' that is preferred over A (i.e. $A'' \preceq A$) and S'' is flagged as a goal state.

Proof. We have already shown that the algorithm without pruning and conflicting actions is complete in a stronger sense (finding *all* non-redundant solutions). We therefore only need to show that pruning and applying conflicting actions can only result in missing a solution state S' when another solution state S'' that is more preferred (i.e. is the result of a more preferred action set) is found.

Let us first consider pruning (lines 4, 17–18, 20–21 in Algorithm 2). We need to show that whenever the algorithm prunes a solution, there exists a more preferred solution that is flagged as a goal state. We observe that the pruning mechanism in lines 17–18 applies only when a goal state is found, and that it operates by pruning goal states that are *less* preferred to the current goal state. This satisfies the desired condition: a goal state is marked as pruned exactly when a more preferred goal state has just been found and marked as a goal.

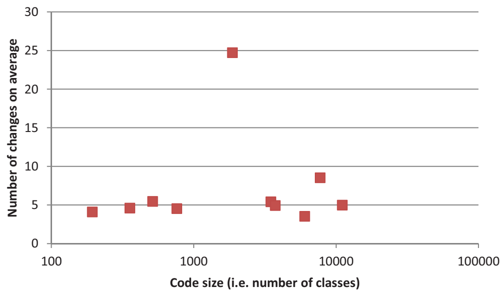
By contrast the pruning in lines 20–21 applies when a constraint is fixed, but other constraints remain (so the current state is not a goal state). We need to argue that pruning does not cause us to miss out on goal states.

Consider a situation where A_1 is a non-redundant solution to a constraint C_1 w.r.t. state S_0, and we are pruning another state $S_1^a = S_0 + A'$, because $A_1 \sqsubseteq A'$ (line 21). By the definition of $\sqsubseteq$, we can therefore view A' as being $A_1 \cup A_{nc}$ for some non-empty set of non-conflicting actions A_{nc}. For pruning to result in a solution not being found, where it would otherwise be found, it must be the case that A' is a solution for C_1 w.r.t. S_0, otherwise the condition in line 19 is false, and the algorithm does not continue to explore from S_1^a. We also must have that there exists A_{2-n} which is a solution for the remaining constraints w.r.t. S_1^a (and it must also keep C_1 true). This is illustrated in the figure below.

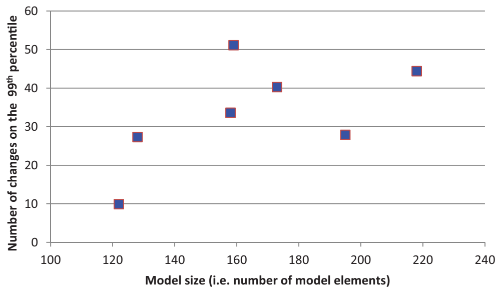
$$\begin{tikzcd}
S_0 \arrow[r, "A_1"] \arrow[d, "A' = A_1 \cup A_{nc}"'] & S_1 \vDash C_1 \arrow[d, "A_{2-n} \cup A_{nc}"] \\
S_1^a \arrow[r, "A_{2-n}"'] & S_2^a \vDash C_1 \ldots C_n
\end{tikzcd}$$

Now, consider applying $A_{2-n} \cup A_{nc}$ to S_1 (right-side vertical arrow in the figure). The accumulated action set applied to S_0 is A_1 (yielding S_1) and then $A_{2-n} \cup A_{nc}$, i.e. $A_1 \cup A_{2-n} \cup A_{nc}$. But this is exactly the same as the action set applied on the pruned path (i.e. $A_1 \cup A_{nc}$ and then A_{2-n}), so we have constructed an alternative path to S_2^a via S_1, rather than via S_1^a. We only need to show that $A_{2-n} \cup A_{nc}$, or a non-redundant subset of it, will be found as a solution for $C_2 \ldots C_n$ w.r.t S_1. We do this by applying Theorem 2 (possibly repeatedly, working backward). Note that if there is a non-redundant subset of $A_{2-n} \cup A_{nc}$, then we need to argue that it is preferred to $A_{2-n} \cup A_{nc}$. We observe that if any of the actions omitted from the non-redundant subset are conflicting actions, then the search process will extend the solution by adding them back in at the end of the search process (lines 9–16). We then focus on non-conflicting actions where, by definition of $\preceq$, the non-redundant subset is preferred to the redundant solution $A_1 \cup A_{2-n} \cup A_{nc}$.

We now turn to applying conflicting actions (lines 9–16). Since this is only applied at the end of a search (when a goal state is reached) it does not reduce the search space, but rather, takes a solution and extends it by considering whether further conflicting actions can be added to it. This can potentially yield a more preferred solution, and so when a solution that has just been found is marked as being dominated (line 15) it is exactly in the situation where a more preferred solution has just been found, and marked as a goal state (in line 14). Similarly, we observe that line 16, which applies when a goal



(a) Java projects



(b) UML models

Fig. 12. The number of changes N between two versions of open-source software projects.

state has been found, only prunes out less preferred potential conflicting action sets, and hence meets the completeness condition. □

6.2. Scalability

The key question is to what extent the computational complexity of the search algorithm scales with the size of the model. The worst-case complexity of the search algorithm is $O(B^D)$ where B is the average branching factor (the maximum number of successors from a state), and D is the maximum depth of the search tree. The branching factor B is the number of combinations from the available actions (derived from the available action set Θ), which in the worst case is 2^N (number of k-combinations for all k) where N is the number of actions in Θ (i.e. the size of Θ). The worse-case depth of the tree is the number of non-persistent inconsistencies (I) plus the number of inconsistencies that are introduced by the repair process (R, for "ripple"), thus $D = I + R$. While exponential growth poses serious *theoretical* threats to our algorithm, we will show, using empirical evidence, that in practice both B and D are very small and more importantly do not increase with the model size. Firstly, we have shown in Egyed (2006), based on an empirical study of 29 UML models (26 of them were third-party models) ranging from small models to very large ones (with hundreds of thousands of model elements), that each (single) change affects a *fixed* (on average 10) number of constraints. Therefore, the number of inconsistencies I in the initial merged model is proportional to the number of changes N rather than to the size of the model, i.e. $I = O(N)$. Secondly, we know from Groher and Egyed (2010) that the ripple effect R in practice is limited to 4 constraint instances and stays constant with the model size. Consequently, $D = O(N + 4)$, i.e. D is proportional to the number of changes N in the available action set Θ.

We thus empirically measured the number of changes in the UML design models of 7 open-source systems (e.g. UseCaseReservation, AnneeEtude, Seance, Services, Formation, UE, Semestre). For each system, we iterated over its version control history to compute N. We found that *99% of all commits involved changes to fewer than 52 model elements* (see Fig. 12(b)). In addition, Fig. 12(b) also shows that the number of changes does not increase with the model size. Since the size of the models we studied was however comparatively small (due to very limited availability of versioned models in open source projects), we performed a similar study on 10 open-source Java systems, which ranged from small to large scale with up to 10,000 classes, e.g. JHotDraw, Log4J, XStream, JDownloader, Psm, Joda_time, Gwt, Tvbrowser, Ant_main, and JEdit. We found that roughly 5 classes (except for the case of the XStream project which has on average 25 classes) were committed on average, but most importantly the number of changes N did not increase with the system size (see Fig. 12(a)). Therefore, evidences from both of the studies on UML models and code suggested that both $B = O(2^N)$ (the exponential base) and $D = O(N)$ (the exponential factor) are also constant. Since both B and D are constant, the *practical* computational complexity of our approach is constant with model size.

In order to complement the theoretical scalability discussion above with a practical empirical assessment of the approach, we have run our tool against four *industrial* UML models[11] (all of which have class, sequence and state diagrams) ranging from small (290 model elements) to large models (33,357 model elements) using 18 consistency constraints (producing from 92 to 13,504 constraint instances). The models are part of the model collections used in Egyed (2011). The evaluations were done using our implementation for the IBM RSA on an Intel Core 2 Quad CPU @2.83 GHz with 8 GB (4 GB available for the RSA) RAM and 64bit Linux (3.1.9). Since we did not have available large UML models with multiple versions and different diagram types, for the purpose of a scalability assessment, we have created the versions of each UML model by randomly introducing percentages of changes relative to the size of the model (because we want to assess our algorithm's performance relative to the number/percentage of changes in addition to model size). This is a worst case assessment since we already know that commit sizes are comparatively small and do not increase with the size of the model.

For each of the four selected UML models, we have done the following. First, we generated a set of random changes from the model. Although our algorithm can handle conflicting changes, in this experiment we assumed that the changes are orthogonal since our focus was on assessing scalability with respect to the number of changes and the model size. Specifically, since conflicting changes are applied at the end of a search, when a goal state is found, the existence of conflicting changes does not affect the "shape" of the search space. We then applied the set of random changes to the model, yielding an initial merged model which was input to our tool. For each action in the set of changes, we obtained its reverse action. The set of all reverse actions form the available action set that was input to our tool. We have also tested our approach with different sizes of Θ, i.e. from 3 up to 1650 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.

The results again show very clearly that the time taken does not increase as the model size increases (e.g. it took only 7 ms for 325 changes in the Insurance Network Fees model with 16,255 model elements). The results also show, as we expected, that the time taken grows with the number of changes (relative to the model size) seeded and the number of non-persistent inconsistencies in the initial merged model (see Figs. 13 and 14; both graphs are in the logarithmic scale). However, it does demonstrate that our approach scales quite well to both a large percentage of changes (e.g. 40% changes in the Microwave Oven model took only 26 ms) and the number of inconsistencies (e.g. 5% changes causing 60 inconsistencies in the biggest[12] ⟨*unnamed*⟩ model

[11] The models are Microwave Oven (with 290 model elements and 92 constraint instances), iTalks (2212 model elements, 1587 constraint instances), Insurance Network Fees (16,255 model elements, 7482 constraint instances), and ⟨unnamed⟩ (33,347 model elements, 13,504 constraint instances).

[12] Name redacted for commercial reasons.

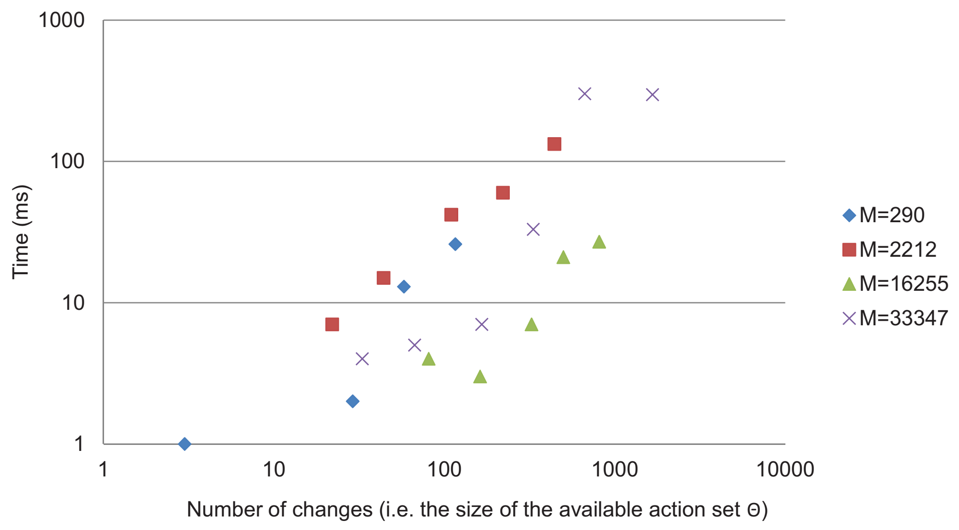


Fig. 13. Computation time for different models and changes (original algorithm). Both *X*-axis and *Y*-axis are in the logarithmic scale.

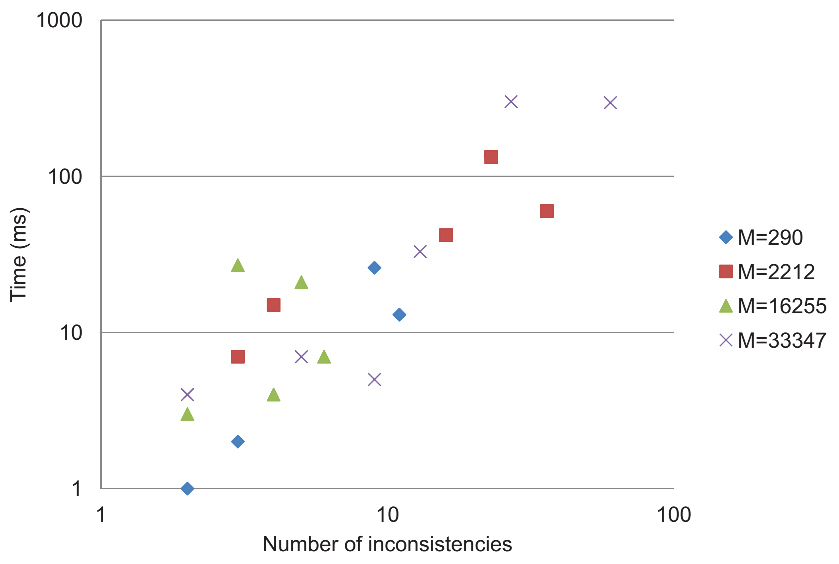


Fig. 14. Computation time for different models and inconsistencies (original algorithm). Both *X*-axis and *Y*-axis are in the logarithmic scale.

of 33,347 model elements took less than 0.3 s). More importantly, our approach performs very well (i.e. took 15 ms or less) in 99% of all classical merging situations where there are fewer than 52 changes. This observation suggests that our approach is able to merge models even with larger differences (1000+ changes). Since models rarely go through dramatic changes, this implies that our approach should scale to nearly all merging scenarios—all but the most extreme cases (e.g., a larger refactoring), which are rare.

We have also conducted the same experiments with the modified algorithm which uses the repair generation. Fig. 15 shows the computing time for all four models against the number of change actions in Θ (noting that both graphs in Figs. 15 and 16 are in the logarithmic scale). The first observation in this result is that the computing time increases with the number of change actions in Θ across all four models. 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 1650 changes to be merged, it took our tool less than 17 s to find all 9 possible solutions for resolving all 71 inconsistencies in the initial merged model.

Fig. 16 shows the computing time against the number of non-persistent inconsistencies in the initial merged model. As can be seen, our approach can quickly find solutions to resolve a large number of inconsistencies: it took less than 9 s to find all six 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 532 ms to find them all, whereas there were only 11 solutions for fixing two inconsistencies, which took 55 ms.

6.3. Memory consumption

Although our algorithm involves looking ahead, it follows a depth-first-search manner and thus memory use is efficient (i.e. linear with respect to the search domain). In addition, we do not store multiple copies of the model in the look ahead. When we want to explore if a change action leads to a solution, we apply the action to the current model, and when we continue to explore an alternative action, the previous action is reversed. In other words, the application of actions is done in a simulated mode, and is undone later in the backtracking mode (not shown in the algorithm), therefore only one copy of the

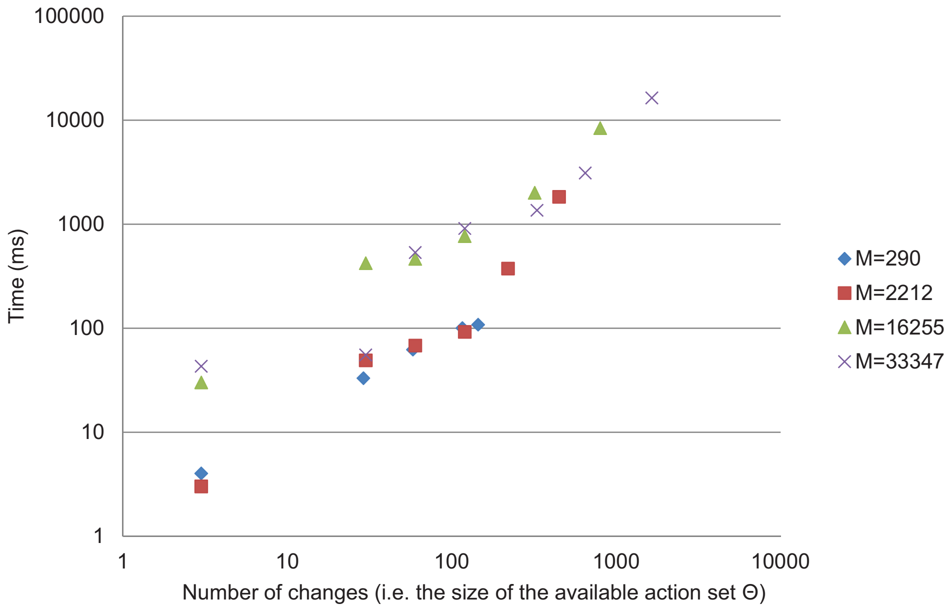


Fig. 15. Computation time for different models and changes (modified algorithm with repair generation). Both *X*-axis and *Y*-axis are in the logarithmic scale .

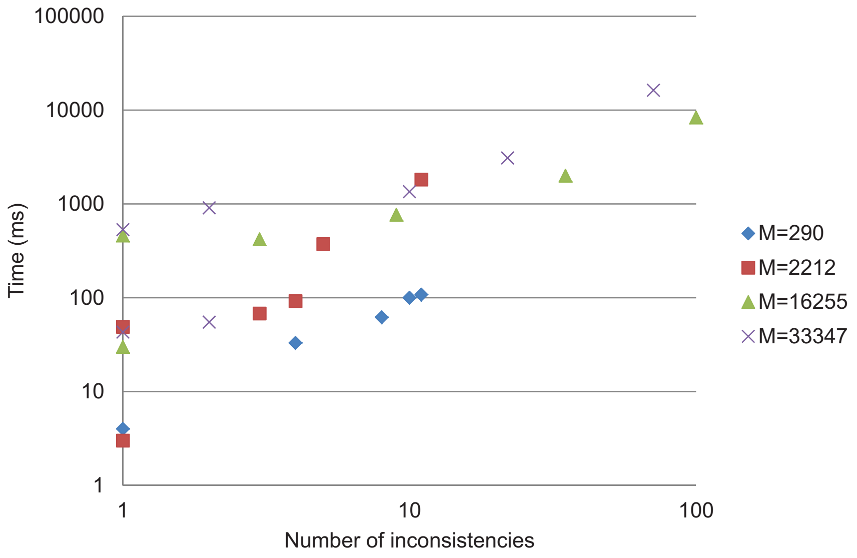


Fig. 16. Computation time for different models and inconsistencies (modified algorithm with repair generation). Both *X*-axis and *Y*-axis are in the logarithmic scale.

model is needed. This technique minimizes the memory usage of our approach and keeps it linear to the depth of the search tree.

6.4. Threats to validity

Our empirical study was performed in the context of four third-party models with vastly different sizes and domains. We have also seeded our evaluation with different numbers of changes between model versions, e.g. from a very small number of 3 changes up to 1665 changes, and with different percentage of changes relative to the model size, e.g. from 0.1% to 40%. However, we acknowledge that those changes which were used to create the model versions were not real changes. We however expect that real versioned changes would give a similar result since our evaluation has covered a wide range of realistic numbers of changes and number of inconsistencies caused by them. A threat to external validity, however, is that we did not yet assess the effectiveness and usability of our approach by monitoring and interviewing software engineers that used our tool. Future investigation would evaluate how difficult it is for users to manually resolve inconsistencies that arise during merging compared to using our tool.

7. Related work

There have been a range of techniques and tools proposed for differencing and merging models. For example, the work in Chen et al. (2004) focuses on identifying the changes between versions of a product line architecture and merging those changes to create a consolidated version. The approach in Abi-Antoun et al. (2006) 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 versioning, which has recently attracted increasing interest from both academia and industry (see Brosch et al., 2012b for a recent survey and the online bibliography compiling an extensive list of relevant publications in this field, Bibliography on Comparison and Versioning of Software Models, 2014). The most important challenges in this field are identifying the changes between model versions (model differencing) and merging those changes to create a consolidated version (model merging) (Rubin and Chechik, 2013). Model differencing techniques are typically classified into state-based approaches and change-based approaches. State-based approaches (e.g. Xing and Stroulia, 2005) calculate the changes between model versions by matching and difference algorithms, while change-based approaches (e.g. Herrmannsdoerfer and Koegel, 2010) record the changes directly in the modeling tool. Some of those advanced techniques (e.g. Xing and Stroulia, 2005) can even identify model differences based on name and structure similarity and do not rely on the assumption of UUID preservation. In addition, there have been several approaches (e.g. Gils, 2002; Maoz et al., 2011a; 2011b) to model matching and differencing which consider the semantics of the models (instead of only the concrete or the abstract syntax). Semantic differencing is just another dimension; both state-based and operation-based can be used for semantic differencing. Our approach is able to utilize either technique to compute the available action set.

Most model merging techniques focus on dealing with conflicts and are classified into two major approaches. The first approach is completely shifting the responsibility of resolving conflicts to the users (e.g. IBM RSA, IBM, 2013 or EMF Compare, Eclipse, 2013a) or guiding the user through the resolution process and suggesting possible resolution strategies (e.g. Gerth et al., 2013). The second approach (e.g. Cicchetti et al., 2008) attempts to automate the conflict resolution process by defining merge policies (e.g. stating which side should be preferred in the merge process). In some cases, it is not necessary for software engineers to resolve merge conflicts immediately. The recent work in Wieland et al. (2013) supports deferring the resolution by annotating conflicts. This prompts the involved parties to discuss and agree on a consolidate decision. Our approach also supports this notion of conflict tolerance by accommodating conflicts in our exploration for possible merging options. However, there has been little work on dealing with inconsistencies in model merging (Brosch et al., 2012b), and most of them (e.g. Bartelt, 2008; Brosch et al., 2012a; Sabetzadeh et al., 2010; Taentzer et al., 2010; Westfechtel, 2010 or the TReMer+ framework, Sabetzadeh et al., 2008, 2007) only deal with identifying inconsistencies in the merged version, not resolving them. Recently, Taentzer et al. (2012) has attempted to address the resolution of inconsistencies (referred to as state-based conflicts in their work) in model merging. However, their work is limited to providing only a highly abstract repair model based on graph modifications and a few examples of repair actions. The recent Eclipse's EMF Diff/Merge incubation project (Eclipse, 2013b), aims to provide consistent merging versions of an EMF model. Their approach is not described in detail but seems to incrementally build the consistent merged model from the differences (instead of having an initial merged model as in our approach) 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. The recent work (Dam et al., 2014) describes our initial attempt to address inconsistencies arisen in merging architectural model using a repair generator for inconsistencies. This paper examines the issues in a more thorough manner and proposes a different approach using search-based techniques.

Recent research has also started to explore continuous merging to identify and help resolve conflicts early when changes to source code are made concurrently by different developers. Some approaches (e.g. Brun et al., 2013; Guimarães and Silva, 2012) create a shadow repository and continuously merge uncommitted and committed changes in the background which is then compiled and tested to detect merge conflicts. Other approaches (e.g. Kasi and Sarma, 2013) focus on determining conflicting tasks and schedule them to recommend conflict-free development paths. Collaborative development environments (e.g. the CoDesign framework, Bang et al., 2010) have also been proposed to allow developers to work in a shared environment, which enables conflict detection in real time. Although those approaches provide valuable solutions to deal with conflicts in collaborative software engineering, it should be noted that conflicts cannot be entirely eliminated in practice. Developers sometimes need to work offline or may not able to work in a co-design manner, or sometimes they want to make a certain amount of changes before considering merging. Recent research (e.g. Goeminne and Mens, 2013) has also explored to determine which identities in a collaborative development environment (e.g. open source software repositories) represent the same physical person and how to merge these identities.

There has been a range of recent work on resolving inconsistencies in models. Some of them, including our prior work (e.g. Egyed, 2007; Reder and Egyed, 2012), only considered fixing single inconsistencies, whereas our work in this paper considers fixing a number of inconsistencies at the same time using information in model versions. Other work also aims to automate inconsistency resolution by having predefined resolution rules (e.g. Liu et al., 2002) or identifying specific change propagation rules for all types of changes (e.g. Briand et al., 2006). Such rules can be formally defined following a logic-based approach (e.g. Liu et al., 2002 used Java Rule Engine JESS, or Mens et al., 2005 used Description Logic, or a graph-based approach such as the graph transformations used in Mens et al., 2006). However, these approaches suffer from the correctness and completeness issue since the rules are developed manually by the user. As a result, there is no guarantee that these rules are complete (i.e. that there are no inconsistency resolutions other than those defined by the rules) and correct (i.e. any of the resolutions can actually fix a corresponding inconsistency). In addition, a significant effort is required to manually hardcode such rules when the number of consistency constraints increases or changes. Our other earlier work (Dam and Winikoff, 2010, 2011, 2008) considers multiple constraints and supports the automated generation of repairs but does not scale well to large models, as opposed to the approach presented in this paper. Some existing approaches (e.g. da Silva et al., 2010) address this scalability issue by limiting the depth of the search tree, thus sacrificing the ability to fix all inconsistencies. 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. Maintaining consistency across models developed in different languages has also attracted attention. For example, the work in Eramo et al. (2012) proposes an approach to propagate changes across models written in different architectural description languages. Leveraging such an approach for merging heterogenous models (i.e. different metamodels) would be part of our future study.

Exploring consistent states of a model also resembles a planning problem, and it can be viewed as such (as in Dam and Winikoff, 2011 or Pinna Puissant et al., 2015) although a planner with pruning and loop detection is needed to deal with a large, potentially infinite, search space. The current problem could be formulated as a Constraint Satisfaction Problem (Hentenryck and Saraswat, 1996). CSPs address the combinatorial problem in which given a set of constraints among variables and a set of (domain) values the variables can take, what choices best satisfy these constraints. However, the formulating process of mapping UML concepts (e.g. classes, associations, attributes, etc.) to CSP's variables and their domain values is complicated and challenging, especially maintaining the traceability between the original UML model and its corresponding CSP. In addition, since the entire UML model may need to be converted, this approach likely will not scale well. Furthermore, while a CSP constraint typically identifies the variables explicitly, an OCL constraint does not identify variables directly but instead gives navigation instructions for the UML model.

Finally, our proposal of exploring model merging alternatives to compute the best merging solutions is related to search-based software engineering (Harman, 2007). A recent work (Kessentini et al., 2013) in this domain has applied genetic algorithm (Koza, 1992) to search for the optimal merging sequence that maximizes the number of successfully applied changes (including conflicting changes) in model merging. However, their work does not consider inconsistency issues and is limited to merging structural models (e.g. class diagrams), which is more suitable to changes driven by model refactoring.

8. Conclusions and future work

This paper presented a novel approach for resolving syntactical and semantic inconsistencies in the merging of model versions. Our approach focuses on not only dealing with conflicts, which most existing work focuses on, but also detecting and avoiding syntactic and semantic inconsistencies arising in the merging process. We have proposed a search-based technique, which systematically explores model merging alternatives to compute the best merging solutions in terms of preserving consistency and integrating as much information in the models to be merged as possible. Our approach is able to find all possible solutions, which resolve all non-persistent inconsistencies introduced, by merging different versions of an model. Our approach also provides guidance for resolving persistent inconsistencies, which pre-exist in the model, in terms of telling the software engineers exactly which model elements should be changed to resolve them. Preprocessing can be applied to detect persistent and non-persistent inconsistencies. Our approach however does not force the designers to resolve all inconsistencies in the models before they can be merged. In fact, we allow inconsistent models to be merged, following the principle of tolerating inconsistencies (Balzer, 1991). 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.

Future work would involve investigating how different consistent merged models can be presented to the user in a user-friendly manner, and exploring how our approach can be applied to other areas related to model merging. Future work would also involve evaluating our approach with real versions of UML models when they become available to us. We have not evaluated our approach and tool with human users to fully assess its effectiveness and usability, which is also part of our future work. In addition, an interesting topic for future work is to apply our approach to merging between design models and source code (e.g. UML diagrams and Java code). In a broad sense, all of the software artifacts (including source code and design) are part of a model describing the software system. Each part represents a view of the model and the semantic understanding of one part of the model can arise from other parts of the model. For example, the meaning of a piece of code involving the interaction between a number of objects can be interpreted in terms of a UML sequence diagram. Hence, our approach can be extended to support merging design and source code provided that semantic constraints that link and extract semantics of different parts of a system can be established. Our future work would also investigate whether it is possible that modification patterns of code and of UML models are similar. Finally, our approach could also be extended to support selective undoing of changes in a model where the designer decides which model elements to undo and our approach automatically suggests related changes that should also be rolled back. In this context, our approach could be used to explore what the other model changes are needed to preserve consistency and this study is thus also part of our future work.

Acknowledgments

Hoa Khanh Dam would like to thank Johannes Kepler University (JKU) for granting him a Visiting Research Fellowship Grant which supported his visit at JKU. Alexander Egyed, Alexander Reder, and Roberto E. Lopez-Herrejon acknowledge that this work was supported by the Austrian Science Fund (FWF): P 25289-N15.

References

Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., Garlan, D., 2006. Differencing and merging of architectural views. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 47–58.

Balzer, R., 1991. Tolerating inconsistency. In: Proceedings of the 13th International Conference on Software Engineering (ICSE'91). IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 158–165.

Bang, J.Y., Popescu, D., Edwards, G., Medvidovic, N., Kulkarni, N., Rama, G.M., Padmanabhuni, S., 2010. Codesign: a highly extensible collaborative software modeling framework. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering, vol. 2. ACM, New York, NY, USA, pp. 243–246.

Bartelt, C., 2008. Consistency preserving model merge in collaborative development processes. In: Proceedings of the 2008 ICSE Workshop on Comparison and Versioning of Software Models. ACM, New York, NY, USA, pp. 13–18.

Bibliography on Comparison and Versioning of Software Models, 2014. http://pi.informatik.uni-siegen.de/CVSM/ (accessed 21 March 2014).

Blanc, X., Mounier, I., Mougenot, A., Mens, T., 2008. Detecting model inconsistency through operation-based model construction. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (Eds.), ICSE. ACM, pp. 511–520.

Briand, L.C., Labiche, Y., O'Sullivan, L., Sowka, M.M., 2006. Automated impact analysis of UML models. J. Syst. Softw. 79 (3), 339–352. http://dx.doi.org/10.1016/j.jss.2005.05.001.

Brosch, P., Egly, U., Gabmeyer, S., Kappel, G., Seidl, M., Tompits, H., Widl, M., Wimmer, M., 2012a. 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. Springer-Verlag, Berlin/Heidelberg, pp. 246–256.

Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M., 2012b. An introduction to model versioning. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (Eds.), Formal Methods for Model-Driven Engineering. Lecture Notes in Computer Science, vol. 7320. Springer, pp. 336–398. (Eingeladen; Vortrag: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinoro, Italy. 2012-06-18-2012-06-23.

Brun, Y., Holmes, R., Ernst, M.D., Notkin, D., 2013. Early detection of collaboration conflicts and risks. IEEE Trans. Softw. Eng. 39 (10), 1358–1375.

Chen, P., Critchlow, M., Garg, A., Westhuizen, C., Hoek, A., 2004. Differencing and merging within an evolving product line architecture. In: Linden, F. (Ed.), Software Product-Family Engineering. Lecture Notes in Computer Science, vol. 3014. Springer, Berlin/Heidelberg, pp. 269–281.

Cicchetti, A., Ruscio, D., Pierantonio, A., 2008. Managing model conflicts in distributed development. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems. Springer-Verlag, Berlin/Heidelberg, pp. 311–325.

Dam, H.K., Reder, A., Egyed, A., 2014. Inconsistency resolution in merging versions of architectural models. In: Proceedings of the 11th IEEE/IFIP Conference on Software Architecture. IEEE, Washington, DC, USA, pp. 153–162.

Dam, H.K., Winikoff, M., 2010. Supporting change propagation in UML models. In: Proceedings of the 26th IEEE International Conference on Software Maintenance. IEEE Computer Society, Washington, DC, USA, pp. 1–10.

Dam, H.K., Winikoff, M., 2011. An agent-oriented approach to change propagation in software maintenance. J. Autonomous Agents Multi-Agent Syst. 23 (3), 384–452. doi:10.1007/s10458-010-9163-0.

Dam, K.H., Winikoff, M., 2008. Cost-based BDI plan selection for change propagation. In: Padgham, L., Parkes, D.C., Mueller, J.P., Parsons, S. (Eds.), Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'2008), Estoril, Portugal, pp. 217–224.

Eclipse, 2013a. EMF Compare. http://www.eclipse.org/emf/compare/ (accessed 11 March 2013).

Eclipse, 2013b. EMF Diff/Merge. http://www.eclipse.org/diffmerge/ (accessed 11 March 2013).

Egyed, A., 2006. Instant consistency checking for the UML. In: Proceedings of the 28th International Conference on Software Engineering, pp. 381–390.

Egyed, A., 2007. Fixing inconsistencies in UML models. In: ICSE'07: Proceedings of the 29th International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 292–301.

Egyed, A., 2011. Automatically detecting and tracking inconsistencies in software design models. IEEE Trans. Softw. Eng. 37 (2), 188–204.

Eramo, R., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A., 2012. A model-driven approach to automate the propagation of changes among architecture description languages.. Softw. Syst. Model. 11 (1), 29–53.

Feiler, P.H., Gluch, D.P., Hudak, J.J., 2006. The architecture analysis & design language (AADL): an introduction. Technical Report. Software Engineering Institute, Carnegie Mellon University.

Gerth, C., Küster, J.M., Luckey, M., Engels, G., 2013. Detection and resolution of conflicting change operations in version management of process models. Softw. Syst. Model. 12 (3), 517–535.

Gils, B.v., 2002. Application of semantic matching in enterprise application integration. Tilburg University, The Netherlands, EU.

Goeminne, M., Mens, T., 2013. A comparison of identity merge algorithms for software repositories. Sci. Comput. Program. 78 (8), 971–986.

Groher, I., Egyed, A., 2010. Selective and consistent undoing of model changes. In: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems. Springer-Verlag, Berlin, Heidelberg, pp. 123–137.

Guimarães, M.L., Silva, A.R., 2012. Improving early detection of software merge conflicts. In: Proceedings of the 2012 International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 342–352.

Harman, M., 2007. The current state and future of search based software engineering. In: 2007 Future of Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 342–357. doi:10.1109/FOSE.2007.29.

Hentenryck, P.V., Saraswat, V., 1996. Strategic directions in constraint programming. ACM Comput. Surv. 28 (4), 701–726. http://doi.acm.org/10.1145/242223.242279

Herrmannsdoerfer, M., Koegel, M., 2010. Towards a generic operation recorder for model evolution. In: Proceedings of the 1st International Workshop on Model Comparison in Practice. ACM, New York, NY, USA, pp. 76–81.

IBM, 2013. IBM Rational Software Architect. http://www.ibm.com/software/rational/products/swarchitect/ (accessed 11 March 2013).

Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B., Silva, J.R.O., 2004. Documenting component and connector views with UML 2.0. Technical Report. Software Engineering Institute (Carnegie Mellon University).

Kasi, B.K., Sarma, A., 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 732–741.

Kessentini, M., Werda, W., Langer, P., Wimmer, M., 2013. Search-based model merging. In: Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference. ACM, New York, NY, USA, pp. 1453–1460.

Koza, J.R., 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA.

Lallchandani, J.T., Mall, R., 2011. A dynamic slicing technique for UML architectural models. IEEE Trans. Softw. Eng. 37 (6), 737–771.

Liu, W., Easterbrook, S., Mylopoulos, J., 2002. Rule based detection of inconsistency in UML models. In: Kuzniarz, L., Reggio, G., Sourrouille, J.L., Huzar, Z. (Eds.), UML 2002, Model Engineering, Concepts and Tools. Workshop on Consistency Problems in UML-based Software Development. Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Ronneby, pp. 106–123.

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A., 2013. What industry needs from architectural languages: A survey. IEEE Trans. Softw. Eng. 39 (6), 869–891.

Maoz, S., Ringert, J.O., Rumpe, B., 2011a. ADDiff: semantic differencing for activity diagrams. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 179–189. doi:10.1145/2025113.2025140.

Maoz, S., Ringert, J.O., Rumpe, B., 2011b. CDDiff: semantic differencing for class diagrams. In: Proceedings of the 25th European Conference on Object-Oriented Programming. Springer-Verlag, Berlin/Heidelberg, pp. 230–254.

Mens, T., 2002. A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. 28 (5), 449–462.

Mens, T., Van Der Straeten, R., D'Hondt, M., 2006. Detecting and resolving model inconsistencies using transformation dependency analysis. In: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems. Springer-Verlag, Berlin/Heidelberg, pp. 200–214.

Mens, T., Van Der Straeten, R., Simmonds, J., 2005. A framework for managing consistency of evolving UML models. In: Yang, H. (Ed.), Software Evolution with UML and XML. Idea Group Publishing, pp. 1–31.

Object Management Group, 2004. UML 2.0 Superstructure and Infrastructure Specifications. http://www.omg.org/technology/uml/.

Pinna Puissant, J., Van Der Straeten, R., Mens, T., 2015. Resolving model inconsistencies using automated regression planning. Softw. Syst. Model. 14 (1), 461–481.

Reder, A., Egyed, A., 2010. 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. ACM, New York, NY, USA, pp. 347–348.

Reder, A., Egyed, A., 2012. Computing repair trees for resolving inconsistencies in design models. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, pp. 220–229.

Rubin, J., Chechik, M., 2013. N-way model merging. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 301–311.

Sabetzadeh, M., Nejati, S., Chechik, M., Easterbrook, S., 2010. Reasoning about consistency in model merging. In: Egyed, A., Lopez-Herrejon, R., Nuseibeh, B., Botterweck, G., Chechik, M., Hu, Z. (Eds.). 3rd Workshop on Living With Inconsistency in Software Development. CEUR Workshop Proceedings.

Sabetzadeh, M., Nejati, S., Easterbrook, S., Chećhik, M., 2008. Global consistency checking of distributed models with TReMer+. In: Proceedings of the 30th International Conference on Software Engineering. ACM, New York, NY, USA, pp. 815–818.

Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., Chechik, M., 2007. Consistency checking of conceptual models via model merging. In: Proceedings of the 15th IEEE International Requirements Engineering Conference, pp. 221–230.

da Silva, M.A.A., Mougenot, A., Blanc, X., Bendraou, R., 2010. Towards automated inconsistency handling in design models. In: Proceedings of the 22nd international conference on Advanced Information Systems Engineering. Springer-Verlag, Berlin/Heidelberg, pp. 348–362.

Taentzer, G., Ermel, C., Langer, P., Wimmer, M., 2010. Conflict detection for model versioning based on graph modifications. In: Proceedings of the 5th International Conference on Graph Transformations. Springer-Verlag, Berlin/Heidelberg, pp. 171–186.

Taentzer, G., Ermel, C., Langer, P., Wimmer, M., 2012. A fundamental approach to model versioning based on graph modifications: from theory to implementation. Softw. Syst. Model. 1–33. doi:10.1007/s10270-012-0248-x.

Westfechtel, B., 2010. A formal approach to three-way merging of emf models. In: Proceedings of the 1st International Workshop on Model Comparison in Practice. ACM, New York, NY, USA, pp. 31–41. doi:10.1145/1826147.1826155.

Wieland, K., Langer, P., Seidl, M., Wimmer, M., Kappel, G., 2013. Turning conflicts into collaboration. Comput. Support. Coop. Work 22 (2–3), 181–240.

Xing, Z., Stroulia, E., 2005. UMLDiff: an algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, pp. 54–65.

Hoa Khanh Dam is a Senior Lecturer at the School of Computing and Information Technology, University of Wollongong, Australia. He holds a Ph.D. degree in Computer Science from RMIT University, Australia. His research has been published in the top venues in AI/intelligent agents (AAMAS, JAAMAS), software engineering (ICSE, ICSM, ER, JSS), and service-oriented computing (ICSOC, SCC, BPM). His work has also won multiple Best Paper Awards (at WICSA, APCCM, and ASWEC) and ACM SIGSOFT Distinguished Paper Award (at MSR).

Alexander Egyed is a Full Professor and Chair for Software-Intensive Systems at the Johannes Kepler University, Austria. He received a Doctorate degree from the University of Southern California, USA. He was a post doc at the University College London, UK, and worked in industry for many years. He is most recognized for his work on software/systems modeling particularly on variability, consistency, and traceability. Dr. Egyed has published over 160 refereed scientific books, journals, and conferences with over 4000 citations to date. He was recognized as a top 1% scholar by the ACM and named an IBM Research Faculty Fellow.

Michael Winikoff's research interests concern notations for specifying and constructing software. In particular, he is interested in agent-oriented software engineering methodologies and is co-author of the book Developing Intelligent Agent Systems: A Practical Guide, published by John Wiley and Sons in 2004. Michael is head of the department of Information Science, at the University of Otago. Before moving to New Zealand he was a member of the agents group at RMIT university's School of Computer Science and IT.

Alexander Reder received his master's degree in software engineering and Ph.D. degree in computer science from the Johannes Kepler University Linz in Austria, in 2009 and 2013 respectively. He is now working at voestalpine group-IT GmbH, Austria. His research interests include model driven engineering and consistency management in model based development.

Roberto E. Lopez-Herrejon is currently a postdoctoral researcher at the Johannes Kepler University in Linz Austria. He has been a Lise Meitner Fellow (2012–2014) sponsored by the Austrian Science Fund (FWF), an Intra-European Marie Curie Fellow (2012–2014) sponsored by the European Union, and a Career Development Fellow (2005–2008) at the Software Engineering Centre of the University of Oxford, England. He obtained his Ph.D. from The University of Texas at Austin in 2006, funded in part by a Fulbright Fellowship. His expertise is software product lines, variability management, feature oriented software development, and search-based software engineering.