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)

http://dx.doi.org/10.1016/j.jss.2015.06.044
0164-1212/

[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.

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.

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.

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.

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*