Property-Based Methods for Collaborative Model Development

Marsha Chechik[1], Fabiano Dalpiaz[3], Csaba Debreceni[4], Jennifer Horkoff[2], István Ráth[4], Rick Salay[1], and Daniel Varró[4]

[1] Department of Computer Science, University of Toronto, Canada
[2] City University London, UK
[3] Utrecht University, the Netherlands
[4] Budapest University of Technology and Economics, Hungary

Abstract. Large-scale software projects are often faced with the challenge of enabling the high degree of collaborative and concurrent development required to meet the aggressive delivery schedules while still maintaining a high standard of system correctness and safety. While Model-Driven Engineering (MDE) can be effective in accelerating development, traditional approaches to addressing the above challenges are designed for code, and are not directly applicable to models. In this paper, we propose a novel approach to addressing the problem of model-based collaborative development using *property-based* techniques. We illustrate our proposals and outline the challenges to realizing them.

1 Introduction

Large-scale software industries are often faced with the challenge of enabling collaborative and concurrent development required to meet aggressive delivery schedules, while maintaining a high standard of system correctness and safety.

In recent years, Model-Driven Engineering (MDE) techniques are being increasingly used to accelerate software development in these domains. Here, models are used as the primary development artifacts amenable to continuous formal verification and validation. Concurrent modeling is becoming increasingly relevant, due to the growing diffusion of cloud-based modeling environments such as LucidChart, draw.io, Gliffy, and Creately.

Effective collaborative development requires the ability to answer several key analysis questions, such as **(Q1)** How to prevent interference between teams potentially making updates to the same portion of the system model? **(Q2)** How to ensure that local changes do not cause global inconsistencies? **(Q3)** How to provide views of the system that are relevant to teams?

In this paper, we propose a set of novel *property-based* strategies for addressing the concurrent model development questions. Question **Q1** is addressed by introducing *property locking* – a technique in which, instead of "hard locking" an entire region of the model, prevents changes introduced by one team that violate the designated properties of other teams. For **Q2**, we propose to enable continuous checking of models w.r.t. *approximate properties* that are quick to check during the model editing time. They are intended to approximate the more detailed checks done later in the verification stage, and are enabled by incremental

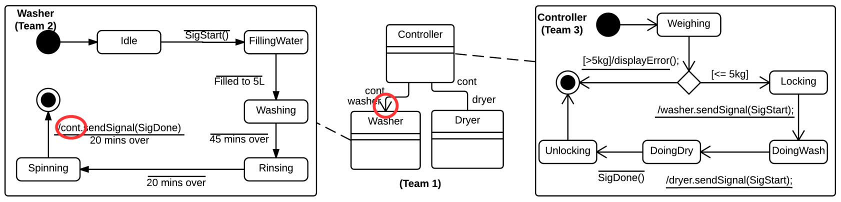

Fig. 1. Washer/Dryer example adapted from [12].

checking as models get changed or when commits into a central repository are attempted. To address **Q3**, teams can define and use overlapping and continuously updated *property views* of the model that show the parts relevant to the properties of interest for resolving a conflicting change.

The property-based approaches we propose are specifically developed for models rather than adapted from techniques for collaborative code development. For code artifacts, these questions have traditionally been addressed by partitioning the code and assigning portions to different teams using file-level locking and textual merging techniques [11] followed by testing/verification procedures such as integration testing or model checking.

Unfortunately, such traditional approaches for managing concurrent *code* development do not naturally extend to concurrent *model-driven* development. Partitioning into fixed model fragments is difficult due to the interconnected, graph-like nature of models. Fixed fragments are inflexible when faced with varying modeling tasks. *Conflict avoidance* techniques such as locking lead to over-locking due to the high degree of interdependence between parts of a model. This significantly limits the degree of concurrent development and does not scale with the increasing number of collaborating teams. *Model merging and conflict detection* can be complex tasks, relying on comparing graphs instead of strings, and the interdependence within a model makes conflicts easy to introduce and hard to resolve. Finally, some model verification and validation techniques are too complex to be executed frequently, making quality control an expensive afterthought. Paul Dourish' pioneering work [5] argues against the inflexibility of locking mechanisms based on the syntax of a collaborative artifact (here, a model) and instead proposes a promise-guarantee paradigm. We are inspired by this approach and bring it to the field of software/system modeling, where the collaborative artifact is a (hyper)graph.

Paper organization. In Sec. 2, we describe a simple collaborative modeling scenario. In Sec. 3, 4 and 5, we describe the three proposed property-based techniques. We conclude in Sec. 6 with a summary and ideas for future work.

2 Motivating Example

We motivate the need for property views, property locking and approximate properties using a simple, multi-model example adapted from [12]. Here, a UML class diagram captures Controller, Washer and Dryer classes (middle of Fig. 1).

The Controller and Washer are further described via statecharts (left and right of Fig. 1). Imagine the case where a number of teams must collaborate in order to modify and update the models. Teams must manage concurrent changes and avoid frequent conflicts. Consider Scenario 1, where Team 1 checks out the class diagram from the repository, while Team 2 checks out the Washer state machine. Now as highlighted by red circles in Fig. 1, Team 1 modifies the class diagram by restricting the navigability of the association cont from the class Washer to Controller. However, as a side-effect of this change, SigDone can no longer be sent from the Spinning state in the Washer (circled in Fig. 1), disrupting the work of Team 2.

Motivated by this scenario, we ask instances of the broader questions from Sec. 1: **q1**: How to prevent an interference when a change made by Team 1 affects the work of Team 2? **q2**: How to immediately detect whether Team 1's change causes an inconsistency with Team 2, without interrupting their development process by executing a potentially slow and costly check? **q3**: How to highlight the parts of the class and state diagrams which are relevant to the assumptions behind Team 2's work, in order to prevent an interference like the one in Scenario 1, or to better understand such interferences when they occur?

Our proposal allows teams to specify what other teams must respect using *properties*. If a property of Team 2 is violated by a change of Team 1 then the change is detected and disallowed (**Q1**). To reduce the number of conflicts, such property violations among collaborative teams need to be detected efficiently upon each change. In case properties of the model are complex to check, e.g., those requiring the use of an external tool such as a model checker, we can use *syntactic approximations* to postpone complex semantic checks (**Q2**). Finally, (**Q3**) we can use properties to derive views, showing collaborators what parts of the model may affect desired properties. The rest of these paper outlines the specificities of this technique using our example.

3 Property Locking

Property locking generalizes the traditional lock-based approach to managing concurrent changes to a model. Instead of having each team lock a model fragment for which they get exclusive write access, teams specify *a property on model elements* that no other team can violate. Such a property captures the assumptions/pre-requisites that the team's changes rely upon. Thus, other teams can make whatever changes they like as long as they do not invalidate these assumptions. The benefit is greater flexibility in allowing concurrent changes.

The properties to lock are propagated to the modeling environment of other collaborating teams and are checked either when a commit is initiated (*batch checks*), or continuously upon each model change (*incremental checks*, e.g., using IncQuery [14]). While property locks are typically specified within the localized context of a few model elements, to allow higher flexibility in collaborative modeling, traditional global properties, e.g., deadlock avoidance, are also possible.

Illustration. To prevent Scenario 1 by maintaining a bidirectional communication between the Washer and Controller classes, Team 2 states property P_1: "each association end of the class diagram that is used in the Washer state machine should remain navigable", and use it as the lock. Thus, when Team 1 attempts to change the navigability of association end cont in the class diagram and commits the change, a violation of property P_1 is detected and the corresponding commit is prevented. Locking using P_1 is less restrictive than traditional model locks which would prevent changes in larger chunks of the model. With property locking, any association end of the class diagram can be changed unless it is used in the state machine Washer. Furthermore, the validity of P_1 can be checked incrementally, so as to detect violations as soon as they occur.

Related work. Dourish' work [5] (see Sec. 1) is most closely related to our proposal. Existing collaborative modeling tools either lack locking support or implement rigid strategies such as file-based locking, or locking subtrees or elements of a specific type, which hinder effective collaboration. Most of *offline collaborative modeling tools*, e.g., ModelCVS [8] or EMFStore [7], rely on traditional version control systems, with contributors committing large deltas of work, resulting in frequent conflicts. *Model repositories* such as CDO[1] and Morsa [6] support locking of subtrees and sets of model elements. These locks can prevent others from accessing elements included in the lock, which weakens scalability w.r.t. the number of collaborators. *Online collaborative modelling frameworks* such as WebGME [10] and ATOMPM [13] rely on a short transaction model: a single, shared instance of the model is concurrently edited by multiple users, with all changes propagated to all participants instantaneously. These systems lack conflict management or provide only lightweight mechanisms, e.g., explicit locking.

Challenges. The crucial problem of our approach is the effective specification of properties for locking. While teams may specify the properties of their interest using a property (or query) language, we believe that the key to property specification is creating (or using) a library of properties which include, for example, well-formedness constraints and design rules for a given modeling language, as well as their customizations for the team's goals and the specific application domain. A secondary challenge concerns the efficient management of property locks with the increasing number of collaborators.

4 Approximate Properties

Formal verification or testing often use semantic properties to check correctness of changes made, after they have been completed. If these checks can be partially moved to the modification time, this could eliminate the check-fail-fix iterations thereby reducing the development time.

To do so, we aim to produce *syntactic* model properties that are efficient to check and that *approximate* the desired semantic properties. Let P be a semantic property that needs to hold. One possibility is to define an approximate property P' to be a "necessary condition" for P, i.e., $\neg P' \Rightarrow \neg P$. Then, if P' is violated,

[1] http://eclipse.org/cdo

there is an error in the model and the change should be rejected. Of course, if P' is satisfied, the original property P should still be checked at a later phase.

Another approximate property type can be used to generate warnings, i.e., a violation of a syntactic property P'' indicates an increased likelihood that the original property P is violated, but does not guarantee it.

Illustration. We use our Washer/Dryer example to illustrate the usage of both types of approximate properties.

Consider a scenario where Team 3 aims to maintain a liveness property of Controller P_2 : "after state Locking, eventually state Unlocking will be reached". We define an approximation P'_2: "for all signals S that are triggers on a path from Locking to Unlocking, there exists some state machine that sends S". Clearly, $\mathsf{P}_2 \Rightarrow \mathsf{P}'_2$ and so satisfying P'_2 is necessary for satisfying P2. Checking P_2 requires a model checker, but P'_2 specifies an efficient incremental syntactic check which helps coordinate the interaction between concurrent teams. If Team 2 were to remove sending of the SigDone signal from the Washer state machine, the violation of P'_2 would be immediately detected, and the change would be rejected.

Now consider a safety property of Controller P_3: "state DoingDry cannot occur before state DoingWash". We define the approximation property P'_3: "no path from state DoingDry to state DoingWash exists". While it is possible to have an infeasible path between these two states, it is suspicious and should be investigated. Thus, a violation of P'_3 may be a potential violation of P_3.

Related work. Producing approximations of behavioral properties in order to simplify model checking is often done using *abstract interpretation*. This technique involves first abstracting the model itself and then deriving the approximated property from the abstraction. In contrast, we propose using abstractions of the property *directly*, without changing the model on which the property is being checked.

Anti-patterns have been studied for software engineering in general [2] and, to a lesser extent, in the context of models [3]. Instances of these patterns are considered to be red-flags indicating potential problems. These are similar to what we mean by approximate warning properties. Yet anti-patterns are generic and apply in any software whereas we are aiming to generate properties specifically for modeling contexts.

Challenges. The main challenge for realizing approximate properties is in developing algorithms for automatically generating efficiently checkable approximate properties from requirements and the current states of the models. A secondary challenge concerns providing the modeler with useful feedback about property violations in a non-intrusive way.

5 Property Views

In order to limit cognitive complexity, teams want to focus their attention to parts of the model that cause, or help avoid, conflicts. Unfortunately, such related model parts are not always easy to find (or available in the form of a diagram) in the underlying modeling language.

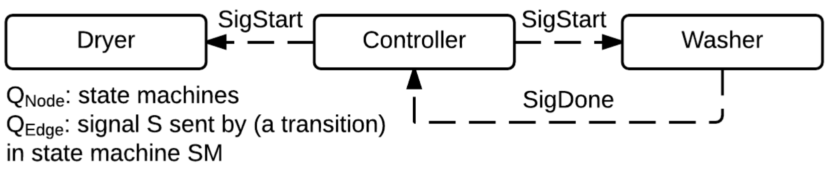

Fig. 2. A property view of a UML model.

We propose to compute property views by adapting *model slicing* [1] to a collaborative scenario. We assume that the slicing criterion is captured by properties to derive a slice by filtering a source model. This model may contain derived elements (i.e., objects, references, attributes) which were not present in the original model but are calculated from it, and it may even comply to a different metamodel.

Illustration. The semantic problem caused by removing the signal SigDone from the Washer state machine is not easy to highlight in UML as it does not have diagrams for depicting the possible event flow between state machines. However, we can use the approximate property P'_2 from Sec. 4 to define a new UML view where each node corresponds to a state machine while each edge between two nodes denotes sending a signal from the source to the target state machine. The corresponding property view, automatically derived by using [4], is shown in Fig. 2.

Related work. [9] identifies four basic approaches used for slicing to support code impact analysis: behavioral – traditional dependency analysis; historical – mining software repositories, e.g., to support co-checkin; textual – measuring coupling, and behavioral execution – collecting execution information. The property view is a fifth approach – finding a subset of the model that impacts satisfaction of the property and thus is relevant for the intended changes.

The use of queries or declarative constraint languages has been explored in [1] for the purpose of computing slices. Slices are frequently defined similarly to views by a declarative query or constraint language. However, unlike slicing, property views need to be maintained *incrementally* upon each change of the model, in order to provide immediate and continuous feedback to collaborating teams. Using the categorization of [1], our approach can be categorized as *active* model slicing with possibly imported output metamodels.

Challenges. Incremental query-based view computation necessitates smart caching and processing of model elements. We intend to rely upon *incremental model query frameworks* [14] for this purpose.

6 Summary and Future Plans

In this paper, we outlined three approaches to using property-based techniques for addressing the problem of model-based collaborative development.

In the future, we will aim to address the challenges identified in the paper. e.g., create approaches for defining and managing property locks, for synthesizing efficiently checkable approximate properties, and for incremental maintenance of property views. We foresee the development of an open source prototype that integrates the concepts of properties with popular collaborative modeling tools.

References

1. A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Kompren: Modeling and Generating Model Slicers. *SoSyM*, 2012.
2. W. Brown, T. McCormick, H.and Mowbray, and R. Malveau. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley, 1998.
3. V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, and C. Trubiani. Digging into UML Models to Remove Performance Antipatterns. In *Proc. of QUOVADIS@ICSE'10*, pages 9–16, 2010.
4. C. Debreceni, Á. Horváth, Á. Hegedüs, Z. Ujhelyi, I. Ráth, and D. Varró. Query-Driven Incremental Synchronization of View Models. In *Proc. of VAO'14*, 2014.
5. P. Dourish. Consistency Guarantees: Exploiting Application Semantics for Consistency Management in a Collaboration Toolkit. In *Proc. of CSCW'96*, 1996.
6. J. Espinazo Pagan, J. Sanchez Cuadrado, and J. García Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *Proc. of MODELS'11*, volume 6981 of *LNCS*. 2011.
7. M. Koegel and J. Helming. EMFStore: A Model Repository for EMF Models. In *Proc. of ICSE'10, Vol. 2*, pages 307–308, 2010.
8. G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Towards a Semantic Infrastructure Supporting Model-Based Tool Integration. In *Proc. of GaMMa@ICSE'06*, 2006.
9. B. Li, X. Sun, H. Leung, and S. Zhang. A Survey of Code-Based Change Impact Analysis Techniques. *J. Software Testing, Verification and Reliability*, 23(8):613–646, 2013.
10. M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendovszky, and A. Lédeczi. Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure. In *Proc. of MPM@MoDELS'14*, pages 41–60, 2014.
11. N. Niu, E. Easterbrook, and M. Sabetzadeh. A Category-Theoretic Approach to Syntactic Software Merging. In *Proc. of ICSM'05*, pages 197–206, 2005.
12. J. Rubin. *Cloned Product Variants: From Ad-Hoc to Well-Managed Software Reuse.* PhD thesis, University of Toronto, 2014.
13. E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. V. Mierlo, and H. Ergin. AToMPM: A Web-based Modeling Environment. In *Proc. of MODELS'13 Tool Demonstrations*, pages 21–25, 2013.
14. Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An Integrated Development Environment for Live Model Queries. *Science of Comp. Prog.*, 2014.