Behavioural Model Fusion: Experiences from Two Telecommunication Case Studies

Shiva Nejati Marsha Chechik
Department of Computer Science, University of Toronto
Toronto, ON M5S 3G4, Canada.
{shiva,chechik}@cs.toronto.edu

ABSTRACT

In large-scale model-based development, developers periodically need to combine collections of interrelated models. These models may capture different features of a system, describe alternative perspectives on a single feature, or express ways in which different features may alter one another's structure or behaviour. We refer to the process of combining a set of interrelated models as *model fusion*. In this position paper, we provide an overview of our work on two key fusion activities, *merging* and *composition*, for behavioural models. The practical basis of our work comes from two case studies that we conducted using models from the telecommunications domain. We illustrate our work using these case studies, summarize the results our research has led to so far, and describe the future research challenges.

Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications.

Keywords: Model Fusion, Model Management, Model Merging, Model Composition, Behavioural Model, Model Matching, Design Patterns, Model Checking.

1. INTRODUCTION

Model-based development involves construction, management, and analysis of complex models. For large-scale projects, this can include several interrelated models, representing different perspectives, different versions across time, different variants in a product family, different components of a system, different development concerns, etc.

The nature of the relationships between a set of models varies based on the intended application of the models and how they were developed. For example, the relationships may describe overlaps (e.g., when the models are different perspectives originating from different sources); or they may describe shared interfaces for interaction (e.g., when the models are autonomous executable components); or they may describe ways in which models alter one another's behaviour or structure (e.g., a cross-cutting model applied to other models). To construct a functional system, models need to be combined with respect to the relationships between them. We refer to this problem as *model fusion*.

Several important activities in model-based development form different facets of model fusion. These include (1) *merging*, used to build a global view of a set of overlapping perspectives (e.g., [20, 16, 24, 22]); (2) *composition*, used to assemble a set of autonomous but interacting components that run sequentially or in parallel (e.g., [8, 10, 12]); and (3) *weaving*, used in aspect-oriented development to incorporate cross-cutting concerns into a base system (e.g., [23]).

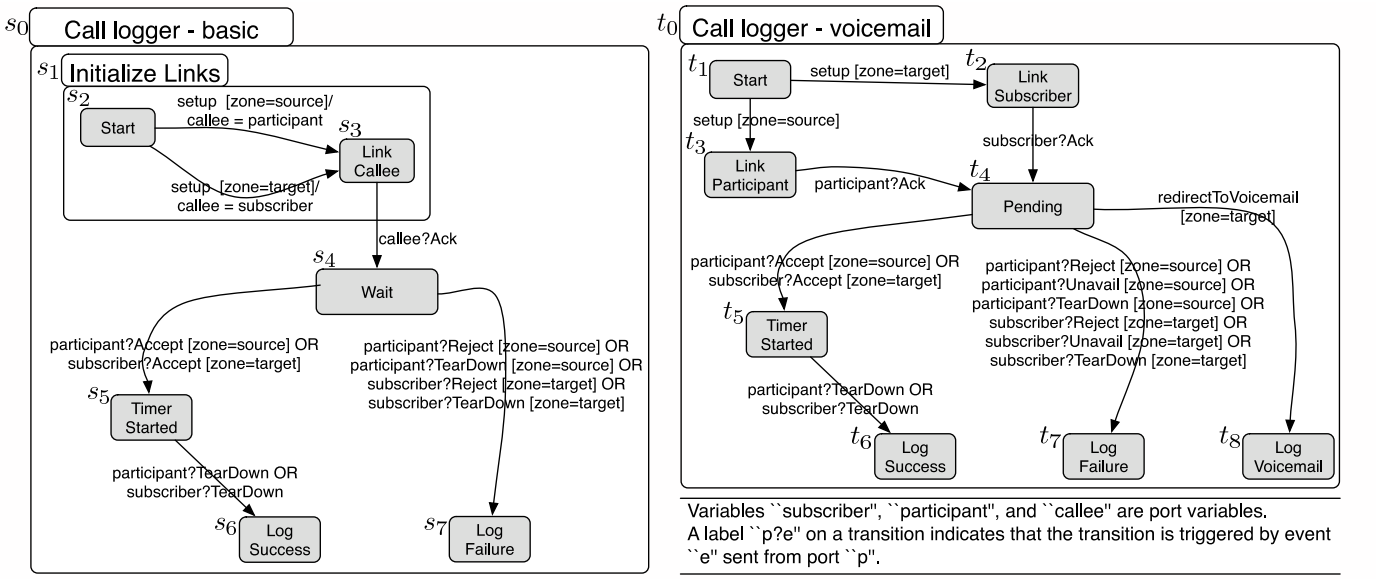
Over the past three years, we have been studying fusion activities in behavioural modelling. Behavioural models capture dynamic aspects of software systems and are described using notations with operational semantics, such as state machines. Our efforts have primarily focused on merging and composition of behavioural models and the automated reasoning tasks that go hand in hand with them (e.g., consistency checking and verification). In this position paper, we provide an overview of the results our research has led to so far, the insights we made along the way, and the challenges we faced, which invite further future research.

The principal source for the observations given here are two case studies that we conducted using models from a telecommunication domain [12]. These case studies, despite dealing with models of the same nature, represent two entirely different fusion problems: The first study concerns maintenance of variant specifications of individual telecom features. The goal here is to merge the variants while preserving their points of difference (i.e., variabilities). In contrast, the goal of the second study is feature interaction analysis. Specifically, given a set of features, we aim to construct a composition of the features which does not exhibit any undesirable behaviours.

In the remainder of this paper, we motivate the two fusion problems (Section 2), summarize our solutions to these problems, and outline the challenges in each case (Sections 3 and 4). We conclude the paper with a plan for future work in Section 5.

2. MODEL FUSION PROBLEMS

One major source for identifying model-based development problems is through studying complex event-driven systems, e.g., automotive or telecom systems. In these systems, a considerable amount of development effort is dedicated to the design of the system architecture and its components. Modelling is used as a suitable vehicle for transforming the high-level design concepts and problem-level ab-



These variants are examples of DFC ``feature boxes'', which can be instantiated in the ``source zone'' or the ``target zone''. Feature boxes instantiated in the source zone apply to all outgoing calls of a customer, and those instantiated in the target zone apply to all their incoming calls. The conditions ``zone = source'' and ``zone = target'' are used for distinguishing the behaviours of feature boxes in different zones.

Figure 1: Simplified variants of the call logger feature.

stractions to software implementations. In our work, we focused on telecom models implemented within the Distributed Feature Composition (DFC) architecture [12], and identified two main fusion problems: merging behavioural models with overlapping behaviours (Section 2.1), and analyzing behavioural models with interacting behaviours (Section 2.2).

2.1 Model Merging

Domain. The motivation for model merging in DFC is to help maintain variant specifications of the *same* feature. For example, Figure 1 shows two variants of the "call logger" feature, aimed to make an external record of the disposition of a call allowing customers to later view information on calls they placed or received. At an abstract level, the feature first tries to setup a connection between the caller and the callee. If for any reason (e.g., caller hanging up or callee not responding), a connection is not established, a failure is logged; otherwise, when the call is completed, information about the call is logged.

Initially, the functionality was designed only for basic phone calls (model *basic* in Figure 1), for which logging is limited to the direction of a call, the address location where a call is answered, success or failure, and the duration if it succeeds. Later, a variant of this feature (model *voicemail* in Figure 1) was developed for customers who subscribe to the voicemail service. Incoming calls for these customers may be redirected to a voicemail resource, and hence, the log information should include the voicemail status as well.

In this domain, telecom features may come in several variants to accommodate different customers' needs. The development of these variants is often distributed across time and over different teams of people, resulting in the construction of independent but *overlapping* models for each feature. For example, the behaviour "After a connection is set up, a successful call will be logged if the subscriber or the participant sends Accept" holds in both models in Figure 1 (through paths from s_4 to s_6, and t_4 to t_6 in basic and voicemail, respectively). This behaviour is a potential overlap between these models.

Goal. To reduce the high costs associated with verifying and maintaining independently developed models, it is desirable to consolidate the variants of each feature into a single coherent model. The main challenge here is to identify correspondences between variant models and merge these models with respect to these correspondences.

2.2 Interaction Analysis

Domain. The second fusion problem concerns the analysis of interactions between concurrent software components. Figure 2 shows the state machines describing two different telecom features: Call Blocking (CB) and Record Voice Mail (RVM). The purpose of CB is to block calling requests coming from addresses on a blocked list. CB becomes active by receiving a setup message containing routing data such as information about the caller and callee. Using this information and its internal logic, CB decides whether the caller should be blocked. If so, it moves to the blocked state and tears down the call; otherwise, it moves to the transparent state and effectively becomes invisible. The purpose of RVM is to record a voice mail message when the callee is not available. Like CB, RVM becomes active by receiving a setup message. It then remains transparent until it receives an unavail message, indicating that the callee is unavailable or is unable to receive the call. Upon the receipt of this message, RVM records a voicemail message through interactions with its media channel, as represented by the internal action

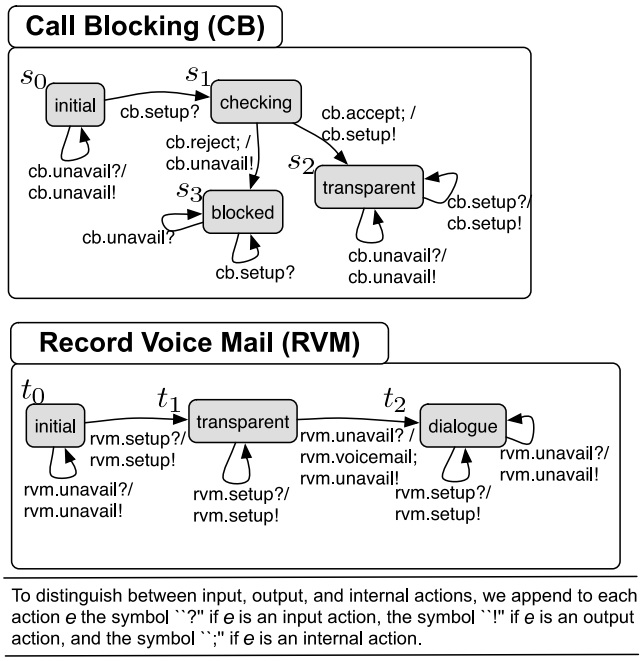


Figure 2: Two different telecom features: Call Blocking (CB); and Record Voice Mail (RVM).

voicemail.

CB and RVM run in synchronous parallel mode and communicate through message passing. They can potentially communicate via their shared messages, i.e., setup and unavail. While the behaviour of each of these components is fixed and deterministic, the behaviour of their parallel composition depends on the messages that are communicated between them. For example, in Figure 3, we have shown two possible compositions of CB and RVM. In the composition in Figure 3(a), RVM sends setup to CB and receives unavail from it, whereas in the composition in Figure 3(b), CB sends setup to RVM and receives unavail from it. The composition in Figure 3(a) results in an undesirable interaction: On the path from (s_0, t_0) to (s_3, t_2), message "rvm.voicemail;" comes after "cb.reject;". That is, we may record a voicemail message from a blocked caller. The composition in Figure 3(b) does not exhibit this undesirable interaction.

Goal. Large systems, such as those from the telecom domain, are often assembled from smaller and modular components. These components, while being independent, interact with one another to perceive and modify the overall function of the system. The major challenge in such systems is orchestrating and managing interactions among components to avoid undesirable behaviours.

3. BEHAVIOURAL MODEL MERGING

3.1 Existing Work

Software engineering deals extensively with model merging – several papers study the subject in specific domains, e.g., use-cases [18], class diagrams [2, 26], and software architectures [1]. These approaches differ in a number of aspects including the notations they support, their algebraic and logical properties, their ability to resolve or tolerate inconsistency, and assumptions they make about the nature of models and their intended use. A preliminary survey of model merging approaches can be found in [19].

In [16], we proposed a framework for merging Statecharts specifications with overlapping behaviours. Our framework has two main components: (1) match, for finding relationships between models, and (2) merge, for combining models with respect to identified relationships. In general, matching is a heuristic process because we can never be completely sure how exactly the models are related. The main challenge in devising a usable match operator is finding a set of effective heuristics that can imitate the reasoning of a domain expert. In our work, we used a number of heuristics including typographic and linguistic similarities between the vocabularies of different models, structural similarities between model elements, and semantic similarities between models based on a quantitative notion of behavioural bisimulation.

In contrast to matching, merging is not heuristic, and is almost entirely automatable. Given a pair of models and a correspondence relation between them, we proposed an automatic behaviour-preserving merge operator [16]. The novelty of this merge procedure is the use of parameterization for representing variabilities between input models: non-shared behaviours in the merged model become guarded. This approach, while allowing us to merge models with behavioural discrepancies, guarantees that the merge preserves, in either guarded or unguarded form, every behaviour of the input models.

For example, Figure 4 shows one potential matching relation between models in Figure 1, and Figure 5 shows the merge of the models of Figure 1 with respect to the matching in Figure 4. In the merge, non-shared transitions are guarded by the boldface conditions representing their source model.

3.2 Challenges

Relationships between models. Since relationships play a crucial role in model-based development, one has to be concerned with the methods for constructing, verifying, and representing these relationships. Developers may find it very hard to identify and manipulate model relationships manually, specially when models are complex, or when the developers are not very familiar with the models. Automatic or semi-automatic match operators, such as the one described in our work, can allow developers to quickly identify appropriate matches with reasonable accuracy.

These operators can be improved in a number of ways. For example, they can be used interactively, with the developer seeding them with some of the more obvious matches, and pruning incorrect ones iteratively. Or, they can be customized for specific domains using learning-based techniques [14].

In addition to identifying model relationships, we need to ensure that these relationships are semantically meaningful. One way to achieve this is to first compute the merge of the given models with respect to their relationships, and then apply automated analyses, e.g., consistency checking, to ensure that the relationships between models result in a merge which satisfies the properties of interest [21]. In situations where merges are very large, we may investigate compo-

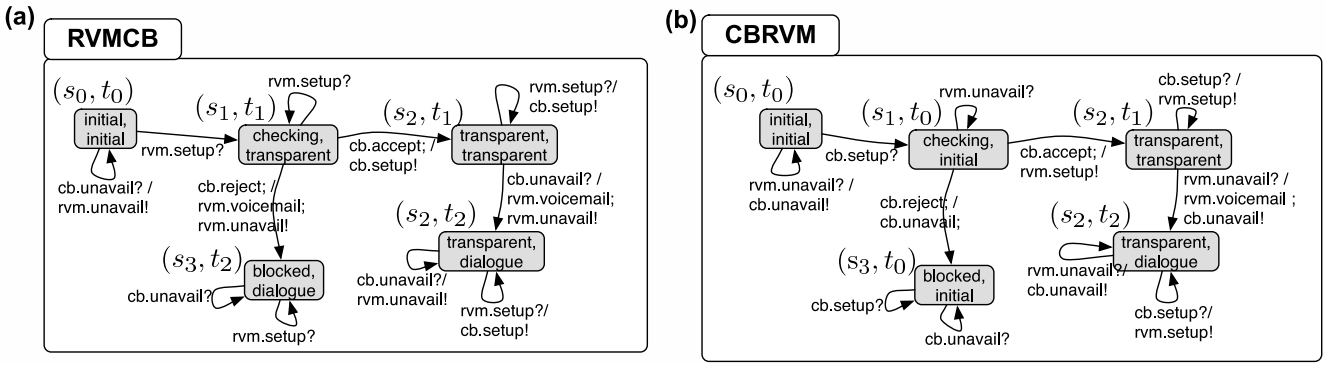


Figure 3: Two possible compositions of the features in Figure 2.

$$(s_0, t_0), (s_4, t_4), (s_2, t_1), (s_5, t_5),$$
$$(s_3, t_2), (s_6, t_6), (s_3, t_3), (s_7, t_7)$$

Figure 4: A correspondence relation between the models in Figure 1.

sitional techniques to reduce reasoning about the merge to reasoning about smaller subsets of models and relationships.

Another significant problem is the *representation* of model relationships. Visual representations are very appealing but they may not scale well for complex operational models such as large executable Statecharts. For these models, it should be possible to express relationships symbolically using logical formulas or regular expressions. This may lead to a more compact and comprehensible representation of model correspondences, especially when tuples of states in a correspondence relation agree on some logical properties or generate similar traces or behaviours.

Heterogeneous merge. Heterogeneous merge is often carried out using transformations and manipulations defined at the meta-model level [5]. Meta-model level transformations, despite being general and flexible, typically deal only with syntactic and visual aspects of models. To generate merges that are semantically sound and to better mechanize the matching process, we could define meta-models that are more than just the abstract syntax of a language, e.g., by augmenting meta-model languages with logical constraints or behavioural specification languages such as activity and sequence models [9].

Tool support. Many industrial distributed and collaborative model-based development tools include some support for model merging. A careful examination of the model merging processes in these tools reveals a number of important shortcomings. In particular, most existing industrial modelling platforms, e.g., the Rational Software Architect [11], are primarily aimed at centralized development, where all developers contribute to a single holistic model. Fragments of this model are visualized as views containing diagrams (potentially) in different notations, e.g., class or sequence diagrams. These tools lack support for merging independently developed models as they often do not allow developers to explicitly construct relationships between

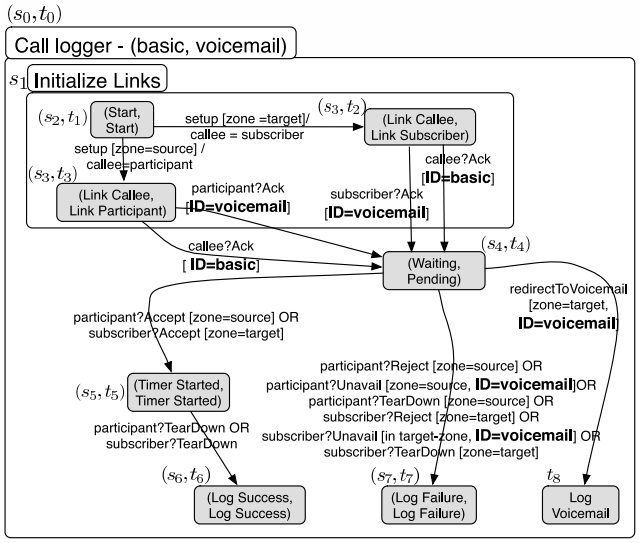


Figure 5: Resulting merge for the call logger variants in Figure 1 with respect to the correspondence relation in Figure 4.

models. In these tools, elements in different views are considered similar only if they are simply different copies of the same element in the holistic model. This is inadequate for merging independently developed models where elements in different models may be similar due to their syntactic and semantic characteristics.

Another issue in the holistic approach to modelling is that even if relationships are made explicit, it is not clear whether they should be defined between models or views. Models usually subsume views in that they contain all the information about the elements of the views. But it would be counter-intuitive for developers to move from a view to its model to specify relationships because models lack the visual layout of views. On the other hand, views may not contain sufficient information for model matching because all information about model elements may not be preserved in the views. Finding the right level of abstraction for defining and representing relationships is an important challenge

in developing model merging tools and designing usable interfaces for these tools.

4. INTERACTION ANALYSIS

4.1 Existing Work

Given a set of interacting components, the goal is to find architectural links (bindings) between them such that the resulting composition does not exhibit any undesirable behaviour. In some sense, this may look like the model merging problem as bindings can be seen as a form of model relationships. However, in distributed component-based development, top-level design decisions about issues such as the overall architecture of a system, component interfaces, and desirable system behaviours are often made prior to the component assembly phase. As a result, finding bindings between interacting components becomes a less heuristic process than finding relationships between the overlapping models. The major issue is that calculating desirable bindings can be very expensive because there are several ways to arrange system components within a given architecture. Analyzing system compositions for all these alternative arrangements takes a lot of time and effort. Hence, we need to provide compositional techniques that can reduce the problem of finding a desirable arrangement of components into smaller subproblems.

A general way to enable such compositional approaches is by exploiting domain-specific patterns used in the design of the components. These patterns often introduce a degree of similarity to the behaviour of the system components, allowing us to construct global system arrangements through the analysis of smaller subsets of components. Specifically, our study of the DFC telecom features shows that DFC components implement a pattern of behaviour known as *transparency*. This pattern requires every component realizing it to have at least one execution path along which it is invisible to other components. The transparent behaviour of feature components is an important factor contributing to their independence and to the freedom with which they can be combined and used in various telecom usages [12].

In [17], we formalize the transparency pattern and propose an automated technique for finding a suitable arrangement of components implementing this pattern, ensuring that undesirable behaviours are not attained. The basic idea behind our technique is that if an ordered pair of components exhibits an undesirable behaviour, then these two components can never appear in this particular ordering even when there is an arbitrary number of components between them. Finding undesirable pairwise orderings between components allows us to prune the search space considerably, and hence, enables efficient computation of a global system arrangement.

4.2 Challenges

Design for verification. To achieve the goal of constructing reliable software systems, we need to promote the use of patterns and guidelines that are able to facilitate efficient automated verification [4, 7]. We conjecture that many existing software design patterns enjoy this characteristic as well: They make software systems more modular and introduce regularity to the behaviour of the system components, enabling efficient compositional verification of system-level properties. To demonstrate this, we need to formalize these patterns and illustrate how they can contribute to more efficient verification of properties of interest. Alternatively, instead of studying general design patterns, we can concentrate on specific problem domains, such as telecom or automotive domains, and let them guide us to the patterns and best practices that have been previously developed by domain experts to make real systems more modular and manageable. By restricting our analysis to specific domains, we may sacrifice claims to generality but we will gain a lot more *credibility* [3].

Verification of a system of models. Fusion tasks are often intertwined with some kind of verification to ensure that the manipulations performed over models preserve their well-formedness and desired semantic properties. For example, in [17], we employ model checking to verify that the composition of a given set of components satisfies the desirable properties. Similarly, [21] combines model merging and (intra-model) consistency checking to enable construction of sound and meaningful relationships between a set of models. To build and manage systems of interrelated models, we need to devise scalable verification techniques that can check not only classical properties of models, but also non-classical ones, such as those involving inconsistent and incomplete aspects of models.

Expressiveness of modelling formalisms. DFC features are implemented in the Boxtalk language, a domain-specific language for specifying telecom components [25]. To analyze these features, we translated them to Input/Output automata [15]. Even though the basis of Boxtalk is very similar to I/O automata, we found a number of domain abstractions in Boxtalk that could not be expressed easily in I/O automata. Domain abstractions are often created to describe complex features of real systems. Lack of support in existing modelling formalisms for capturing them shows a potential gap between real-world systems and the expressive power of these formalisms. To address this gap, we need to design analyzable modelling formalisms that are sufficiently expressive for various domain-specific abstractions. Below, we summarize two of the most interesting problems we faced when formalizing Boxtalk.

- *Determining a model's vocabulary:* The vocabulary of DFC features, i.e., their input and output actions, cannot be determined statically. This is because these features may be instantiated in different telecom pipelines, and the vocabulary of a pipeline depends on the union of vocabulary of its features. For example, feature RVM generates an action loggedVM, indicating that a voicemail message was logged by RVM. The only feature that uses this action, and hence is always enabled for it, is the call logger feature. However, other features, should they appear in between RVM and call logger in a pipeline, also have to be enabled for loggedVM in order to let this message pass through. Yet, these features do not need to be enabled for loggedVM if RVM and call logger do not appear in the pipeline.

  In Boxtalk, a shorthand called *signal-linkage self-loop* is designed to directly connect the right and left ports of a feature, allowing features to pass arbitrary signals form their left neighbour to their right neighbour, and vice versa. Using signal-linkages, Boxtalk models do

not need to make their vocabulary explicit. However, in general-purpose formalisms such as I/O automata, we need to determine the vocabulary of models statically and explicitly label their transitions with appropriate actions from their vocabulary. We may be able to address this problem by assuming that models are incomplete and adopting an open-world approach to modelling.

- *Dynamic bindings:* In DFC, architectural links, i.e., bindings, are dynamic, allowing features to change roles at runtime. Boxtalk represents bindings between features as dynamic *call variables* that can be created, destroyed, and reassigned at runtime. Since I/O automata do not provide any means for describing bindings, we had to extend them as follows: First, we added a notion of action type to I/O automata to explicitly indicate which action belongs to which call variable. We then implemented bindings between I/O automata using a relabeling mechanism [13]. However, we still could not capture dynamic aspects of DFC bindings because our typing and relabeling mechanisms are static. To solve this problem, we need to study the techniques developed for verifying adaptive and self-managing systems [6].

5. CONCLUSION

In this paper, we presented our work on two fusion problems identified in the context of a telecom domain. We outlined a list of challenges that we faced in our work, providing suggestions for future research in this area.

Analyzing a set of models with interacting behaviours is a well-studied problem, e.g., [10, 12]. Existing notions of composition for such models, however, are mostly studied for abstract-level modelling languages such as LTSs or I/O automata. We believe advancement in this area requires focusing on domain specific languages. This would allow us to (1) understand the shortcomings of general purpose modelling formalisms for capturing problem-level abstractions, and (2) identify useful domain specific patterns and guidelines that can facilitate efficient management and analysis of complex component-based systems.

Merging software artifacts is a broader problem, spanning several application domains, from requirements models to code. Even though existing techniques mostly focus on particular applications, they lack sufficient scalability and usability. As a result, adoption by practitioners has been limited. Therefore, we should study the problem of model merging in the context of very large and heterogeneous systems and develop usable tool support for such systems.

Acknowledgments. We would like to thank Mehrdad Sabetzadeh for his insightful comments on earlier drafts of this paper. We are grateful to Steve Easterbrook, Mehrdad Sabetzadeh, Rick Salay, Sebastian Uchitel, and Pamela Zave for many useful discussions and collaboration on various pieces of the research on model fusion.

6. REFERENCES

[1] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan. "Differencing and Merging of Architectural Views". In *Proceedings of ASE'06*, pages 47–58, 2006.

[2] M. Alanen and I. Porres. "Difference and Union of Models". In *Proceedings of UML'03*, pages 2–17, 2003.

[3] T. Ball. "The Verified Software Challenge: A Call for a Holistic Approach to Reliability". In *Proceedings of VSTTE'05*, 2005.

[4] A. Betin-Can, T. Bultan, M. Lindvall, B. Lux, and S. Topp. "Application of Design for Verification with Concurrency Controllers to Air Traffic Control Software". In *Proceedings of ASE'05*, pages 14–23, 2005.

[5] J. Bezivin, F. Jouault, and D. Touzet. "An Introduction to the ATLAS Model Management Architecture". Technical Report 05-01, LINA, 2005.

[6] B. Cheng and J. Atlee. "Research Directions in Requirements Engineering". In *Future of SE Track of ICSE'07*, pages 285–303, 2007.

[7] B. Cheng, R. Stephenson, and B. Berenbach. "Lessons Learned from Automated Analysis of Industrial UML Class Models (An Experience Report)". In *Proceedings of MoDELS'05*, pages 324–338, 2005.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, USA, 1999.

[9] R. France and B. Rumpe. "Model-driven Development of Complex Software: A Research Roadmap". In *Future of SE Track of ICSE'07*, pages 37–55, 2007.

[10] J. Hay and J. Atlee. "Composing Features and Resolving Interactions". In *Proceedings of FSE'00*, pages 110–119, 2000.

[11] IBM Rational Software Architect. `http://www.ibm.com/software/awdtools/architect/swarchitect/`.

[12] M. Jackson and P. Zave. "Distributed Feature Composition: a Virtual Architecture for Telecommunications Services". *IEEE TSE*, 24(10):831 –847, 1998.

[13] J. Magee and J. Kramer. *Concurrency: State models and Java Programming: 2nd Edition*. Wiley, 2006.

[14] D. Mandelin, D. Kimelman, and D. Yellin. "A Bayesian Approach to Diagram Matching with Application to Architectural Models". In *Proceedings of ICSE'06*, pages 222–231, 2006.

[15] S. Nejati. "Translating BoxTalk Models to I/O Automata", 2007. `http://www.cs.toronto.edu/~shiva/boxtalk/`.

[16] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. "Matching and Merging of Statecharts Specifications". In *Proceedings of ICSE'07*, pages 54–64, 2007.

[17] S. Nejati, M. Sabetzadeh, M. Chechik, S. Uchitel, and P. Zave. "Compositional Synthesis of Pipeline Architectures". Submitted, 2008.

[18] D. Richards. "Merging individual conceptual models of requirements". *Requirements Engineering Journal*, 8(4):195–205, 2003.

[19] M. Sabetzadeh. "A Survey of Approaches to Model Merging". Depth Paper, 2006.

[20] M. Sabetzadeh and S. Easterbrook. "View Merging in the Presence of Incompleteness and Inconsistency". *Requirements Engineering Journal*, 11(3):174–193, 2006.

[21] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik. "Consistency Checking of Conceptual Models via Model Merging". In *Proceedings of RE'07*, pages 221–230, 2007.

[22] S. Uchitel and M. Chechik. "Merging Partial Behavioural Models". In *Proceedings of FSE'04*, pages 43–52, 2004.

[23] J. Whittle, A. Moreira, J. Araújo, P. Jayaraman, A. Elkhodary, and R. Rabbi. "An Expressive Aspect Composition Language for UML State Diagrams". In *Proceedings of MoDELS'07*, pages 514–528, 2007.

[24] J. Whittle and J. Schumann. "Generating Statechart Designs from Scenarios". In *Proceedings of ICSE'00*, pages 314–323, 2000.

[25] P. Zave and M. Jackson. "A Call Abstraction for Component Coordination". *Elect. Notes in Theor. CS*, 66(4), 2002.

[26] A. Zito, Z. Diskin, and J. Dingel. "Package Merge in UML 2: Practice vs. Theory?". In *Proceedings of MoDELS'06*, pages 185–199, 2006.