Matching and Merging of Variant Feature Specifications

Shiva Nejati, *Member*, *IEEE Computer Society*,
Mehrdad Sabetzadeh, *Member*, *IEEE Computer Society*,
Marsha Chechik, *Member*, *IEEE Computer Society*,
Steve Easterbrook, *Member*, *IEEE Computer Society*, and
Pamela Zave, *Member*, *IEEE Computer Society*

Abstract—Model Management addresses the problem of managing an evolving collection of models by capturing the relationships between models and providing well-defined operators to manipulate them. In this paper, we describe two such operators for manipulating feature specifications described using hierarchical state machine models: Match, for finding correspondences between models, and Merge, for combining models with respect to known or hypothesized correspondences between them. Our Match operator is heuristic, making use of both static and behavioral properties of the models to improve the accuracy of matching. Our Merge operator preserves the hierarchical structure of the input models, and handles differences in behavior through parameterization. This enables us to automatically construct merges that preserve the semantics of hierarchical state machines. We report on tool support for our Match and Merge operators, and illustrate and evaluate our work by applying these operators to a set of telecommunication features built by AT&T.

Index Terms—Model management, match, merge, hierarchical state machines, statecharts, behavior preservation, variability modeling, parameterization

1 Introduction

Model-based development involves construction, integration, and maintenance of complex models. For large-scale projects, modeling is often a distributed endeavor involving multiple teams at different organizations and geographical locations. These teams build multiple interrelated models, representing different perspectives, different versions across time, different variants in a product family, different development concerns, etc. Identifying and verifying the relationships between these models, managing consistency, propagating change, and integrating the models are major challenges. These challenges are collectively studied under the heading of *Model Management* [1].

Model management aims to provide appropriate constructs for specifying the relationships between models and systematic operators to manipulate the models and their relationships. Such operators include, among others, *Match*, for finding correspondences between models, *Merge*, for putting together a set of models with respect to known relationships between them, *Slice*, for producing a projection of a model based on a given criterion, and *Check-Property*, for verifying models and relationships against the properties of interest [2], [3], [1].

Among these operators, Match and Merge play a central role in supporting distribution and coordination of modeling tasks. In any situation where models are developed independently, Match provides a way to discover the relationships between them, for example, to compare variants [4], to identify inconsistencies [5], to support reuse and refactoring [6], [7], and to enable web-service recovery [8]. Sophisticated Match tools, e.g., Protoplasm [9], can handle models that use different vocabularies and different levels of abstraction. Merge provides a way to gain a unified perspective [10], to understand interactions between models [11], and to perform various types of analysis such as synthesis, verification, and validation [12], [13].

Many existing approaches to model merging concentrate on syntactic and structural aspects of models to identify their relationships and to combine them. For example, Melnik [3] studies matching and merging of conceptual database schemata; Mehra et al. [14] propose a general framework for merging visual design diagrams; Sabetzadeh and Easterbrook [15] describe an algebraic approach for merging requirements viewpoints; and Mandelin et al. [4] provide a technique for matching architecture diagrams using machine learning. These approaches treat models as graphical artifacts while largely ignoring their semantics. This treatment provides generalizable tools that can be applied to many different modeling notations and which are particularly suited to early stages of development, when models may have loose or flexible semantics. However, structural model merging becomes inadequate for later stages of

- *S. Nejati and M. Sabetzadeh are with the SnT Centre, University of Luxembourg, 4 rue Alphonse Weicker, L-2721 Luxembourg. E-mail: {shiva.nejati, mehrdad.sabetzadeh}@uni.lu.*
- *M. Chechik and S. Easterbrook are with the Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, ON M5S 3G4, Canada. E-mail: {chechik, sme}@cs.toronto.edu.*
- *P. Zave is with AT&T Laboratories-Research, 180 Park Ave, Bldg 103, Florham Park, NJ 07932. E-mail: pamela@research.att.com.*

Manuscript received 20 Sept. 2010; revised 25 Oct. 2011; accepted 13 Nov. 2011; published online 17 Nov. 2011.
Recommended for acceptance by D. Giannakopoulou.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-09-0285.
Digital Object Identifier no. 10.1109/TSE.2011.112.

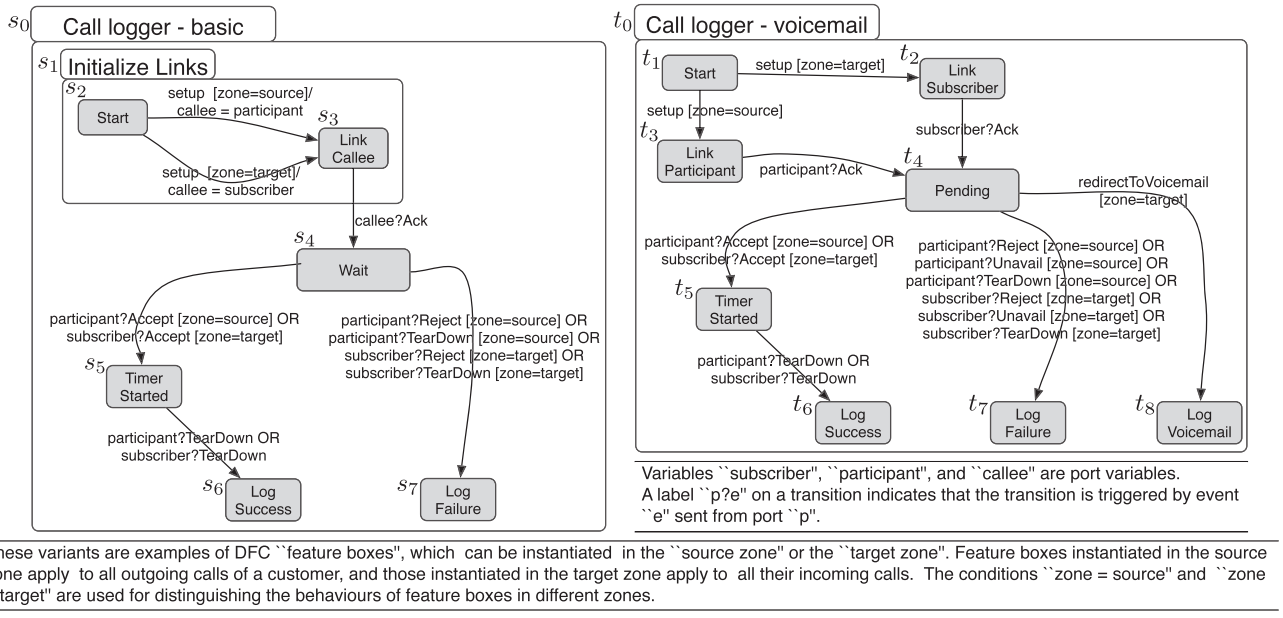

Fig. 1. Simplified variants of the call logger feature.

development where models have rigorous semantics that needs to be preserved in their merge. Furthermore, such an outlook leaves unused a wealth of semantic information that can help better mechanize the identification of relationships between models.

In contrast, research on behavioral models concentrates on establishing semantic relationships between models. For example, Whittle and Shumann [16] use logical pre/post-conditions over object interactions for merging sequence diagrams, and Uchitel and Chechik [12] and Fischbein et al. [13] use refinement relations for merging consistent and partial state machine models so that their behavioral properties are preserved. These approaches, however, do not make the relationships between models explicit and do not provide means for computing and exploring alternative relationships. This can make it difficult for modelers to guide the merge process, particularly when there is uncertainty about how the contents of different models should map onto one another, or when the models are defined at different levels of abstraction.

In this paper, we present an approach to matching and merging variant feature specifications described as Statechart models. Merging combines structural and semantic information present in the Statechart models and ensures that their behavioral properties are preserved. In our work, we separate identification of model relationships from model integration by providing independent Match and Merge operators. Our Match operator includes heuristics for finding terminological, structural, and semantic similarities between models. Our Merge operator parameterizes variabilities between the input models so that their behavioral properties are guaranteed to be preserved in their merge. We report on tool support for our Match and Merge operators, and illustrate and evaluate our work by applying these operators to a set of telecommunication features built by AT&T.

1.1 Motivating Example

Domain. We motivate our work with a scenario for maintaining variant feature specifications at AT&T. These executable specifications are modules within the Distributed Feature Composition (DFC) architecture [17], [18], and form part of a consumer voice-over-IP service [19]. The features are specified as Statecharts.

One feature of the voice-over-IP service is "call logging," which makes 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 works as follows: It first tries to setup a connection between the caller and the callee. If for any reason (e.g., the caller hanging up or the 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, for which logging is limited to the direction of a call, the address location where a call is answered, the success or failure of the call, and the duration if it succeeds. Later, a variant of this feature 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. Fig. 1 shows *simplified* views of the *basic* and *voicemail* variants of this feature. To avoid clutter, we combine transitions that have the same source and target states using the disjunction (OR) operator.

In the DFC architecture, 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

P1	After a connection is set up, a successful call will be logged if the subscriber or the participant sends Accept
P2	After a connection is set up, a voicemail will be logged if the call is redirected to the voicemail service

Fig. 2. Sample behavioral properties of the models in Fig. 1: **P1** represents an overlapping behavior, and **P2** a nonoverlapping one.

feature. For example, consider the two properties described in Fig. 2. Property **P1** holds in both variants shown in Fig. 1 because both can log a successful call: **P1** holds via the path from s_4 to s_6 in the basic variant and via the path from t_4 to t_6 in voicemail. This property represents a potential overlap between the behaviors of these variants. In contrast, property **P2** only holds in voicemail (via the path from t_4 to t_8) but not in basic. This property illustrates a behavioral variation between the variants shown in Fig. 1.

Goal. To reduce the high costs associated with verifying and maintaining independent models, we need to identify correspondences between variant models so that developers can obtain a single unified model.

1.2 Contributions of This Paper

Applications of Match and Merge arise in a number of different contexts, one of which is illustrated by our motivating example. Implementing Match and Merge involves answering several questions. Particularly, what criteria should we use for identifying correspondences between different models? How can we quantify these criteria? How can we construct a merge given a set of models and their correspondences? How can we distinguish between shared and nonshared parts of the input models in their merge? What properties of the input models should be preserved by their merge? In this paper, we address these questions for models expressed as Statecharts. This paper extends and refines an earlier version of this work which appeared in [20], making the following contributions:

- A description of a versatile Match operator for hierarchical state machines. Our Match operator uses a range of heuristics, including typographic and linguistic similarities between the vocabularies of different models, structural similarities between the hierarchical nesting of model elements, and semantic similarities between models based on a quantitative notion of behavioral bisimulation. We apply our Match operator to a set of telecom feature specifications developed by AT&T. Our evaluation indicates that the approach is effective for finding correspondences between real-world models.
- A description of a Merge operator for Statechart models. We provide a procedure for constructing behavior-preserving merges that also respect the hierarchical structuring and parallelism of the input models. We use this Merge operator for combining variant telecom features from AT&T based on the relationships computed by our Match operator between the features.
- Tool support for our Match and Merge operators. Our tool, named TReMer+ (http://se.cs.toronto.edu/index.php/TReMer) [21], enables establishing relationships between models—identified manually or based on results of our Match operator—and computes the result of Merge for each identified relationship.

The rest of this paper is organized as follows: Section 2 provides an overview of our Match and Merge operators. Section 3 outlines our assumptions and fixes notation. Section 4 introduces our Match operator, and Section 5 our Merge operator. Section 6 describes tool support for the two operators. Section 7 presents an evaluation of effectiveness for the Match operator, and Section 8 assesses the soundness of the Merge operator. Section 9 compares our contributions with related work and discusses the results presented in this paper. Finally, Section 10 concludes the paper.

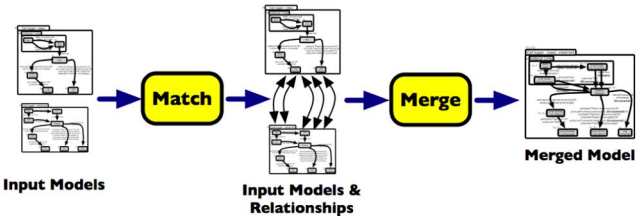

Fig. 3. A framework for integrating variant feature specifications.

2 Overview of Our Approach

Fig. 3 provides an overview of our framework for integrating variant feature specifications. The framework has two main steps. In the first step, a Match operator is used to find relationships between the input models. In the second step, an appropriate Merge operator is used to combine the models with respect to the relationships computed by Match. This framework enables the explicit distinction between the identification of model relationships and model integration—the Match and Merge operators are independent, but they are used synergistically to allow us to hypothesize alternative ways of combining models and to compute the result of merge for each alternative.

Our ultimate goal is to provide automated tool support for the framework in Fig. 3. Among these two operators, Match has a heuristic nature. Since models are developed independently, we can never be entirely sure how these models are related. At best, we can find heuristics that can imitate the reasoning of a domain expert. In our work, we use two types of heuristics: static and behavioral. Static heuristics use structural and textual attributes, such as element names, for measuring similarities. For the models in Fig. 1, static heuristics would suggest a number of good correspondences, e.g., the pairs (s_6, t_6), and (s_7, t_7); however, these heuristics would miss several others, including (s_3, t_3), (s_3, t_2), and (s_4, t_4). These pairs are likely to correspond not because they have similar static characteristics, but because they exhibit similar dynamic behaviors. Our behavioral heuristic can find these pairs.

To obtain a satisfactory correspondence relation, we use a combination of static and behavioral heuristics. Our Match operator produces a correspondence relation between states in the two models. For the models in Fig. 1, it may yield the correspondence relation shown in Fig. 8b. Because the approach is heuristic, the relation must be reviewed by a domain expert and adjusted by adding any missing correspondences and removing any spurious ones. In our example, the final correspondence relation approved by a domain expert is shown in Fig. 8d.

Unlike matching, merging is not heuristic and, in situations where the purpose of merge is clear, this operator is entirely automatable. Given a pair of models and a correspondence relation between them, our Merge operator automatically produces a merge that:

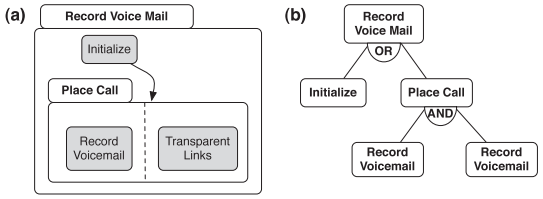

Fig. 4. Parallel Statecharts: (a) An example and (b) the hierarchy tree corresponding to (a).

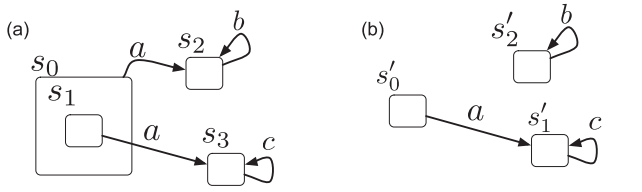

Fig. 5. (a) Prioritizing transitions to eliminate nondeterminism in ECharts: Transition $s_1 \rightarrow s_3$ has higher priority than transition $s_0 \rightarrow s_2$, and (b) the flattened form of the Statecharts in (a).

1. Preserves the behavioral properties of the input models. Fig. 10 shows the merge of the models of Fig. 1 with respect to the relation in Fig. 8d. This merge is behavior preserving. That is, any behavior of the input models is preserved in the merge model (either through shared or nonshared behaviors). For example, the property **P1** in Fig. 2 that shows an overlapping behavior between the models in Fig. 1 is preserved in the merge as a shared behavior (denoted by the path from state (s_4, t_4) to (s_6, t_6)).
2. Distinguishes between shared and nonshared behaviors of the input models by attaching appropriate guard conditions to nonshared transitions. In the merge, nonshared transitions are guarded by boldface conditions representing the models they originate from. For example, the property **P2** in Fig. 2, which holds over the voicemail variant but not over the basic, is represented as a parameterized behavior in the merge (denoted by the transition from (s_4, t_4) to t_8), and is preserved only when its guard holds.
3. Respects the hierarchical structure and parallelism of the input models, providing users with a merge that has the same conceptual structure as the input models.

3 Assumptions and Background

The Statechart language [22] is a common notation for describing hierarchical state machines and is a defacto standard for specifying intra-object behaviors of software systems. Below, the syntax of this language is formalized [23].

Definition 1 (Statecharts). *A Statecharts model is a tuple $(S, \hat{s}, <_h, E, V, R)$, where S is a finite set of states, $\hat{s} \in S$ is an initial state, $<_h$ is an AND-OR tree defining the state hierarchy tree (or hierarchy tree, for short), E is a finite set of events, V is a finite set of variables, and R is a finite set of transitions, each of which is of the form $\langle s, e, c, \alpha, s', prty \rangle$, where $s, s' \in S$ are the transition's source and target, respectively, $e \in E$ is the triggering event, c is an optional predicate over V, α is a sequence of zero or more actions that generate events and assign values to variables in V, and $prty$ is a number denoting the transition's priority.*

We write a transition $\langle s, e, c, \alpha, s', prty \rangle$ as $s \xrightarrow[prty]{e[c]/\alpha} s'$. Each state in S can be either an atomic state or a superstate. A superstate can be an AND-state or an OR-state. The substates of an AND-state are executed in parallel and can be active simultaneously, whereas the substates of an OR-state are executed sequentially and at each time only one state can be active. For example, in Fig. 4a, Record Voice Mail is an OR-state with two substates, Initialize and Place Call, that are executed sequentially, while Place Call is an AND-state with two parallel substates, Record Voicemail and Transparent Links. The hierarchy tree $<_h$ is an AND-OR tree that defines a partial order on states with the top superstate as root and the atomic states as leaves. The hierarchy tree for the model in Fig. 4a is shown in Fig. 4b. In the Statechart of the basic call logger model in Fig. 1, s_0 is the root, s_2 through s_7 are leaves, and s_1 is an OR-state. The set $\hat{s}$ of initial states is $\{s_0, s_1, s_2\}$. The set E of events is {setup, Ack, Accept, Reject, Tear-Down}, and the set V of variables is {callee, zone, participant, subscriber}. The only actions in Fig. 1 are callee = participant and callee = subscriber. These actions assign values participant and subscriber to the variable callee, respectively.

Implementations of the Statechart language differ on how they define the semantics of inter and intramachine communication, and how they resolve nondeterminism in the language [23]. The implementation of the AT&T features is based on a Statechart dialect called ECharts [24] and makes the following choices regarding these issues:

- **Inter and intramachine communication.** ECharts does not permit actions generated by a transition of a Statechart to trigger other transitions of the same Statechart [25]. That is, an external event activates at most one transition, not a chain of transitions. Therefore, notions of macro and microsteps coincide in ECharts.
- **Nondeterminism.** In Statecharts, it may happen that a state and some of its proper descendants have outgoing transitions on the same event and condition, but to different target states. For example, in Fig. 5a, states s_0 and s_1 have transitions labeled a to two different states, s_2 and s_3, respectively. This makes the semantics of this Statechart model nondeterministic because it is not clear which of the transitions, $s_0 \rightarrow s_2$ or $s_1 \rightarrow s_3$, should be taken upon receipt of the event a. In ECharts, certain types of nondeterminism are resolved by assigning global priorities (using $prty$) to transitions that have the same event and condition. For example, in Fig. 5a, it is assumed that the inner transitions have a higher priority than the outer transitions, and hence, on receipt of a, the transition from s_1 to s_3 is activated. The models shown in Fig. 1 are already deterministic, i.e., any external event triggers at most one transition in them. Thus, no further prioritization is required.[1]

1. Note that the ECharts semantics is not fully deterministic. In particular, the ECharts priority rules do not resolve nondeterminism in AND-states.

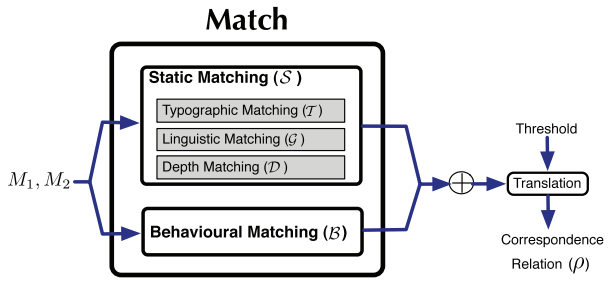

Fig. 6. Overview of the Match operator.

Note that our Matching and Merging techniques are general and can be applied to various Statechart dialects. In order to demonstrate that the Merge operator is semantic preserving, one needs to explicitly identify how the above semantic variation points are resolved in a particular Statechart implementation. Our proof for semantic preservation of Merge (see Appendix XI-C, which can be found in the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2011.112) can carry over to other dialects.

In addition, we make the following assumptions on how behavioral models are developed in our context. Let $M_1 = (S_1, \hat{s}, <_h^1, E_1, V_1, R_1)$ and $M_2 = (S_2, \hat{t}, <_h^2, E_2, V_2, R_2)$ be Statechart models.

- We assume that the sets of events, E_1 and E_2, are drawn from a shared vocabulary, i.e., there are no name clashes and no two elements represent the same concept. This assumption is reasonable for design and implementation models because events and variables capture observable stimuli, and for these, a unified vocabulary is often developed during upstream lifecycle activities. Note that this assumption is also valid for variables in V_1 and V_2 that appear in the guard conditions, i.e., the environmental (input) variables.
- Since M_1 and M_2 describe variant specifications of the *same* feature, they are unlikely to be used together in the same configuration of a system, and hence, unlikely to interact with one another. Therefore, we assume that actions of either M_1 or M_2 cannot trigger events in the other model. For example, the only actions in the Statechart in Fig. 1 are callee => participant and callee = subscriber. These actions do not cause any interaction between the Statechart models in Fig. 1. Hence, the models in Fig. 1 are noninteracting. For a discussion on distinctions between models with interacting versus overlapping behaviors, see Section 9.

4 Matching Statecharts

Our Match operator (Fig. 6) uses a hybrid approach combining static matching, $\mathcal{S}$ (Section 4.1), and behavioral matching, $\mathcal{B}$ (Section 4.2). Static matching is independent of the Statechart semantics and combines typographic and linguistic similarity degrees between state names, respectively, denoted $\mathcal{T}$ and $\mathcal{G}$, with similarity degrees between state depths in the models' hierarchy trees, denoted $\mathcal{D}$. Behavioral matching ($\mathcal{B}$) generates similarity degrees between states based on their behavioral semantics. Each matching is defined as a total function $S_1 \times S_2 \rightarrow [0..1]$, assigning a normalized value to every pair $(s,t) \in S_1 \times S_2$ of states. The closer a degree is to one, the more similar the states s and t are (with respect to the similarity measure being used). We aggregate the static and behavioral heuristics to generate the overall similarity degrees between states (Section 4.3). Given a similarity threshold, we can then determine a correspondence relation ρ over the states of the input models (Section 4.3).

4.1 Static Matching

Static matching, $\mathcal{S}$, is calculated by combining typographic ($\mathcal{T}$), linguistic ($\mathcal{G}$), and depth ($\mathcal{D}$) similarities. In this paper, we use the following formula for the combination:

$$\mathcal{S} = \frac{4 \cdot max(\mathcal{T}, \mathcal{G}) + \mathcal{D}}{5}.$$

The typographic, linguistic, and depth heuristics are described below.

Typographic Matching ($\mathcal{T}$) assigns a value to every pair (s,t) by applying the N-gram algorithm [26] to the name labels of s and t. Given a pair of strings, this algorithm produces a similarity degree based on counting the number of their identical substrings of length N. We use a generic implementation of this algorithm with trigrams (i.e., $\mathrm{N} = 3$). For example, the result of trigram matching for some of the name labels of the states in Fig. 1 is as follows:

$$\begin{aligned} &\mathrm{trigram}(\text{“Wait”}, \text{“Pending”}) &&= 0.0\\ &\mathrm{trigram}(\text{“LogSuccess”}, \text{“LogFailure”}) &&= 0.21\\ &\mathrm{trigram}(\text{“LogSuccess”}, \text{“LogSuccess”}) &&= 1.0\\ &\mathrm{trigram}(\text{“LinkCallee”}, \text{“LinkParticipant”}) &&= 0.18. \end{aligned}$$

Linguistic Matching ($\mathcal{G}$) measures similarity between name labels based on their linguistic correlations, to assign a normalized similarity value to every pair of states. We employ the freely available WordNet::Similarity package [27] for this purpose. WordNet::Similarity provides implementations for a variety of semantic relatedness measures proposed in the Natural Language Processing (NLP) literature. In our work, we use the *gloss vector* measure [28]—an adaptation of the popular cosine similarity measure [29] used in data mining for computing a similarity degree between two words based on the available dictionary and corpus information. For a given pair of words, the gloss vector measure is a normalized value in $[0..1]$.

In many cases, the name labels whose relatedness is being measured are phrases or short sentences, e.g., “Log Success” and “Log Failure” in Fig. 1. In these cases, we need an aggregate measure that computes degrees for name labels expressed as sentences or phrases. To this end, we use a simple measure from natural language processing [26], described below.

We treat each name label as a set of words (which implies that the parts of speech of the words in the name labels are ignored) and compute the gloss vector degrees for all word pairs of the input labels. We then find a matching between the words of the input labels such that the sum of the degrees is maximized. This optimization problem is easily cast into the maximum weighted bipartite graph matching problem, also known as the assignment problem [30]. The

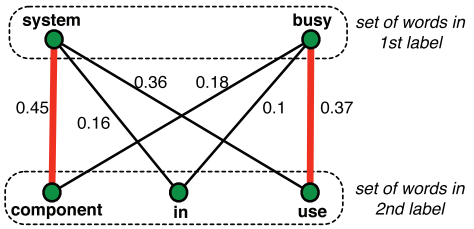

Fig. 7. Weighted bipartite graph induced by a pair of name labels ("system busy" and "component in use").

nodes on each side of the bipartite graph are the words in one of the input labels. There is an edge e with weight w between word x of the first input label and word y of the second input label if the degree of relatedness between x and y is w. The result of maximum weighted bipartite matching is a set of edges $e_1, \ldots, e_k$ such that no two edges have the same node (i.e., word) as an endpoint, and the sum $\sum_{i=1}^{k} weight(e_i)$ is maximal. If the input name labels are for a pair of states (s,t), linguistic similarity between s and t is given by the following:

$$\mathcal{G}(s,t) = \frac{2 \times \sum_{i=1}^{k} weight(e_i)}{N_1 + N_2},$$

where N_1 and N_2 are the number of words in each of the two name labels being compared.

As an example, suppose we want to compute a degree of similarity between the labels "system busy" and "component in use." Fig. 7 shows the weighted bipartite graph induced by the two labels. The weight assigned to each edge denotes the gloss vector degree for the two words connected by the edge. The maximal weight match is achieved when "system" is matched to "component" and "busy" is matched to "use," giving us a match value of $2 \times (0.45 + 0.37)/(2 + 3) \approx 0.33$.

Depth Matching ($\mathcal{D}$) uses state depths to derive a similarity heuristic for models that are at the same level of abstraction. This captures the intuition that states at similar depths are more likely to correspond to each other and is computed as follows:

$$\mathcal{D}(s,t) = 1 - \frac{|depth(s) - depth(t)|}{max(depth(s), depth(t))},$$

where $depth(s)$ and $depth(t)$ are, respectively, the position of states s and t in the state hierarchy tree orderings $<$ of their corresponding input models. For example, in Fig. 1, $depth(s_2)$ is 2 and $depth(t_1)$ is 1, and $\mathcal{D}(s_2, t_1) = 0.5$.

4.2 Behavioral Matching

Behavioral matching ($\mathcal{B}$) provides a measure of similarity between the behaviors of different states. Our behavioral matching technique draws on the notion of bisimilarity between state machines [31]. Bisimilarity provides a natural way to characterize behavioral equivalence. Bisimilarity is a recursive notion and can be defined in a forward and backward way [32]. Two states are *forward bisimilar* if they can transition to (forward) bisimilar states via identically labeled transitions and are (forward) dissimilar otherwise.

Dually, two states are *backward bisimilar* if they can be transitioned to from (backward) bisimilar states via identically labeled transitions and are (backward) dissimilar otherwise.

Bisimilarity relates states with *precisely* the same set of behaviors, but it cannot capture *partial* similarities. For example, states s_4 and t_4 in Fig. 1 transit to (forward) bisimilar states s_7 and t_7, respectively, with transitions labeled participant?Reject[zone=source], participant?TearDown[zone=source], subscriber?Reject[zone=target], and subscriber?TearDown[zone=target]. However, despite their intuitive similarity, s_4 and t_4 are dissimilar because their behaviors differ on a few other transitions, e.g., the one labeled redirectToVoicemail[zone=target].

Instead of considering pairs of states to be either bisimilar or dissimilar, we introduce an algorithm for computing a *quantitative* value measuring how close the behaviors of one state are to those of another. Our algorithm iteratively computes a similarity degree for every pair (s,t) of states by aggregating the similarity degrees between the immediate neighbors of s and those of t. By neighbors, we mean either successor/child states (forward neighbors) or predecessor/parent states (backward neighbors), depending on which bisimilarity notion is being used. The algorithm iterates until either the similarity degrees between all state pairs stabilize or a maximum number of iterations is reached.

In the remainder of this section, we describe the algorithm for the forward case. The backward case is similar. We use the notation $s \xrightarrow{a} s'$ to indicate that s' is a forward neighbor of s. That is, s has a transition to s' labeled a or s' is a child of s where a is a special label called *child*. Note that s can be either an AND-state or an OR-state. Treating children states as neighbors allows us to propagate similarities from children to their parents and vice versa.

We denote by $\mathcal{B}^i(s,t)$ the degree of similarity between states s and t after the ith iteration of the matching algorithm. Initially, all states of the input models are assumed to be bisimilar, so $\mathcal{B}^0(s,t)$ is 1 for every pair (s,t) of states. Users may override the default initial values, for example, assigning zero to those tuples that they believe would not correspond to each other. This enables users to apply their domain expertise during the matching process. Since behavioral matching is iterative, user input gets propagated to all tuples and can hence induce an overall improvement in the results of matching.

For proper aggregation of similarity degrees between states, our behavioral matching requires a measure for comparing transition labels. A transition label is made up of an event and, optionally, a condition and an action. We compare transition labels using the N-gram algorithm augmented with some simple semantic heuristics. This algorithm is suitable because of the assumption that a shared vocabulary for observable stimuli already exists. The algorithm assigns a similarity value $L(a,b)$ in $[0..1]$ to every pair (a,b) of transition labels.

Having described the initialization data ($\mathcal{B}^0$) and transition label comparison (L), we now describe the computation of $\mathcal{B}$. For every pair (s,t) of states, the value of $\mathcal{B}^i(s,t)$ is computed from 1) $\mathcal{B}^{i-1}(s,t)$, 2) similarity degrees between the forward neighbors of s and those of t after step $i-1$, and 3) comparison between the labels of transitions relating s and t to their forward neighbors.

We formalize the computation of $\mathcal{B}^i(s,t)$ as follows: Let $s \xrightarrow{a} s'$. To find the best match for s' among the forward neighbors of t, we need to maximize the value $L(a,b) \times \mathcal{B}^{i-1}(s',t')$, where $t \xrightarrow{b} t'$.

The similarity degrees between the forward neighbors of s and their best matches among the forward neighbors of t after the $i-1$th iteration is computed by

$$X = \sum_{s \xrightarrow{a} s'} max_{t \xrightarrow{b} t'} L(a,b) \times \mathcal{B}^{i-1}(s',t').$$

And the similarity degrees between the forward neighbors of t and their best matches among the forward neighbors of s after iteration $i-1$ are computed by

$$Y = \sum_{t \xrightarrow{a} t'} max_{s \xrightarrow{b} s'} L(a,b) \times \mathcal{B}^{i-1}(s',t').$$

We denote the sum of X and Y by $Sum^i(s,t)$.

The value of $\mathcal{B}^i(s,t)$ is computed by first normalizing $Sum^i(s,t)$ and then computing its average with $\mathcal{B}^{i-1}(s,t)$:

$$\mathcal{B}^i(s,t) = \frac{1}{2}\left(\frac{Sum^i(s,t)}{|succ(s)| + |succ(t)|} + \mathcal{B}^{i-1}(s,t)\right).$$

In the above formula, $|succ(s)|$ and $|succ(t)|$ are the number of forward neighbors of s and t, respectively. The larger the $\mathcal{B}^i(s,t)$, the greater the similarity of the behaviors of s and t is. For backward behavioral matching, we perform the above computation for states s and t, but consider their backward neighbors instead of their forward neighbors.

The above computation is performed iteratively until the difference between $\mathcal{B}^i(s,t)$ and $\mathcal{B}^{i-1}(s,t)$ for all pairs (s,t) becomes less than a fixed $\varepsilon > 0$. If the computation does not converge, the algorithm stops after some predefined maximum number of iterations. Finally, we compute behavioral similarity, $\mathcal{B}$, as the maximum of forward behavioral and backward behavioral matching.

4.3 Combining Similarities and Translating Them to Correspondences

To obtain the overall similarity degrees between states, we need to combine the results from different heuristics. There are several approaches to this, including linear and nonlinear averages and machine learning techniques. Learning-based techniques have been shown to be effective when proper training data are available [4]. Since such data were not present for our case study, our current implementation uses a simple approach based on linear averages. To produce an overall combined measure, denoted $\mathcal{C}$, we take an average of $\mathcal{B}$ with static matching, $\mathcal{S}$ (described in Section 4.1). Fig. 8a illustrates $\mathcal{C}$ for the models in Fig. 1.

To obtain a correspondence relation between input Statechart models M_1 and M_2, the user sets a threshold for translating the overall similarity degrees into a binary relation ρ. Pairs of states with similarity degrees above the threshold are included in ρ and the rest are left out. In our example, if we set the threshold value to 60 percent, we obtain the correspondence relation ρ shown in Fig. 8b.

Since matching is a heuristic process, the resulting binary correspondence relation (ρ) should be reviewed and, if necessary, manually adjusted by the user. For example, in Fig. 1, since states s_6 and s_7 of basic and states t_6, t_7, and t_8 of voicemail do not have any outgoing transitions, there is a high degree of (forward) behavioral similarity between them, and hence, all the pairs (s_6,t_6), (s_6,t_7), (s_6,t_8), (s_7,t_6), (s_7,t_7), and (s_7,t_8) appear in ρ in Fig. 8b. Among these pairs, however, only (s_6,t_6) and (s_7,t_7) are valid correspondences according to the user. We assume the user would remove the rest of the pairs from ρ. As we discuss in Section 5.1, the relation ρ may need to be further revised before merging to ensure that the resulting merged model is well formed.

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
t_0	**.87**	**.63**	.54	.03	.08	.07	.57	.58
t_1	.48	**.70**	**.92**	.17	.17	.26	.20	.23
t_2	.08	.18	.17	**.65**	.30	.31	.31	.29
t_3	.07	.19	.17	**.66**	.30	.32	.30	.30
t_4	.07	.15	.17	.23	**.64**	.30	.30	.30
t_5	.08	.15	.25	.22	.24	**1.0**	.04	.28
t_6	.58	.45	.17	.22	.30	.30	**1.0**	**.63**
t_7	.56	.45	.17	.22	.31	.28	**.62**	**1.0**
t_8	.55	.45	.17	.22	.30	.35	**.62**	**.62**

(a) Combined $\mathcal{C}$ matching results for the models in Figure 1.

$(s_0,t_0),(s_2,t_1),(s_3,t_2),(s_3,t_3),\ (s_4,t_4),(s_5,t_5),(s_6,t_6),$
$(s_7,t_7),(s_1,t_0),(s_1,t_1),(s_6,t_7),(s_6,t_8),(s_7,t_6),(s_7,t_8)$

(b) A correspondence relation ρ.

$(s_0,t_0),(s_2,t_1),(s_3,t_2),(s_3,t_3),\ (s_4,t_4),(s_5,t_5),(s_6,t_6),$
$(s_7,t_7),(s_6,t_7),(s_6,t_8),(s_7,t_6),(s_7,t_8)$

(c) The relationship in (b) after applying sanity checks of Section V-A.

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

(d) The relations in (c) after user revisions.

Fig. 8. Results of matching for call logger.

5 Merging Statecharts

In this section, we describe our Merge operator for Statecharts. The input to this operator is a pair of Statechart models M_1 and M_2, and a correspondence relation ρ. The output is a merged model if ρ satisfies certain sanity checks (discussed in Section 5.1). These checks ensure that merging M_1 and M_2 using ρ results in a well-formed (i.e., structurally sound) Statechart model. If the checks fail, a subset of ρ violating the checks is identified.

5.1 Sanity Checks for Correspondence Relations

To produce structurally sound merges, we need to ensure that ρ passes certain sanity checks before applying the Merge operator:

1. The initial states of the input models should correspond to one another. If ρ does not match $\hat{s}$ to $\hat{t}$, we add to the input models new initial states $\hat{s}'$ and $\hat{t}'$ with transitions to the old ones. We then simply add the tuple $(\hat{s}',\hat{t}')$ to ρ. Note that we can lift the behavioral properties of the models with the old

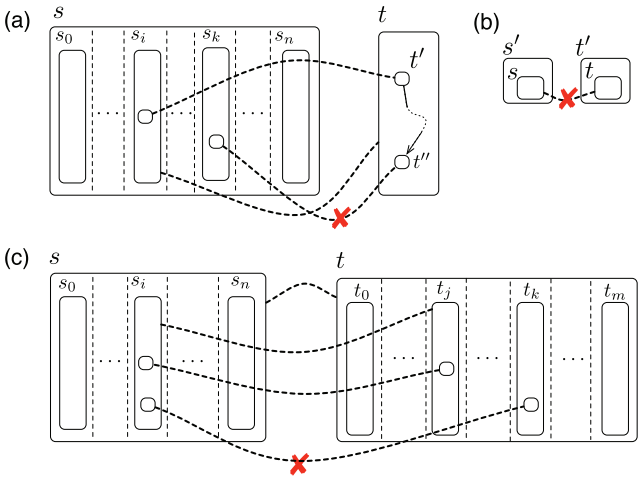

Fig. 9. Example violation of sanity checks: (a) and (c) Violations of AND-states integrity rules and (b) violation of relational adequacy.

initial states to those with the new initial states. For example, instead of evaluating a temporal property p at $\hat{s}$ (respectively, $\hat{t}$), we check AXp at $\hat{s}'$ (respectively, $\hat{t}'$), where AX denotes the universal next-time operator—we borrow it from the commonly used temporal logic CTL [33].

2. The correspondences in ρ must respect the input models' hierarchy trees. That is, ρ must satisfy the following conditions:
   a. *(monotonicity)* For every $(s,t) \in \rho$, if ρ relates a proper descendant of s (respectively, t) to a state x in M_2 (respectively, M_1), then x must be a proper descendant of t (respectively, s).
   b. *(relational adequacy)* For every $(s,t) \in \rho$, either the parent of s is related to an ancestor of t or the parent of t is related to an ancestor of s by ρ.
   c. *(AND-states integrity rules)*
      i. For every $(s,t) \in \rho$, if s (respectively, t) is an AND-state, t (respectively, s) has to be an AND-state as well.
      ii. Let s be an AND-state and let $s_0, \ldots, s_n$ be parallel substates of s. Then, if ρ maps a proper descendant of s_i; $(0 \leq i \leq n)$ to a state t',
         A. there must be some ancestor of t' mapped to s_i by ρ, and
         B. it cannot map a proper descendant of s_k $(0 \leq k \neq i \leq n)$ to t'', where t'' can reach t', or can be reached from t', or is a child or an ancestor of t'. See Fig. 9a.
      iii. Let s and t be AND-states, let $s_0, \ldots, s_n$ be parallel substates of s, and let $t_0, \ldots, t_m$ be parallel substates of t. If ρ maps a proper descendant of s_i $(0 \leq i \leq n)$ to a proper descendant of t_j $(0 \leq j \leq m)$, then 1) it has to map s_i to t_j and s to t, and 2) it cannot map another proper descendant of s_i (respectively, t_j) to a proper descendant of t_k $(0 \leq k \neq j \leq m)$ (respectively, s_k $(0 \leq k \neq i \leq n)$). See Fig. 9c.

Monotonicity ensures that ρ does not relate an ancestor of s to t (respectively, t to s) or to a child thereof. Relational adequacy ensures that ρ does not leave parents of both s and t unmapped; otherwise, it would not be clear which state should be the parent of s and t in the merge. Note that descendant, ancestor, parent, and child are all with respect to each model's hierarchy tree, $<_h$. AND-states integrity rules ensure that ρ never maps an AND-state to a nonparallel state, and further, that it never maps a pair of states in the same parallel region to states in different parallel regions or vice versa. The latter condition is to ensure that the resulting merge never has transitions crossing parallel regions.

Pairs in ρ that violate any of the above conditions are reported to the user. In our example, the relation shown in Fig. 8b has three monotonicity violations: 1) s_0 and its child s_1 are both related to t_0, 2) t_0 and its child t_1 are both related to s_1, and 3) s_1 and its child s_2 are both related to t_1. Our algorithm reports $\{(s_0,t_0),(s_1,t_0)\}$, $\{(s_1,t_0),(s_1,t_1)\}$, and $\{(s_1,t_1),(s_2,t_1)\}$ as conflicting sets. Suppose that the user addresses these conflicts by eliminating (s_1,t_0) and (s_1,t_1) from ρ. The resulting relation, shown in Fig. 8d, passes all sanity checks and can be used for merge. Consider the example shown in Fig. 9b. In this example, states s and t are related but their parents are not. Since it is not possible to have multiple parents for single states in the merged model, this is a violation of the relational adequacy condition.

5.2 Merge Construction

Let M_1 and M_2 be Statechart models. To merge them, we first need to identify their shared and nonshared parts with respect to ρ. A state x is *shared* if it is related to some state by ρ, and is *nonshared* otherwise. A transition $r = \langle x, a, c, \alpha, y, prty \rangle$ is *shared* if x and y are, respectively, related to some x' and y' by ρ, and further, there is a transition r' from x' to y' whose event is a, whose condition is c, whose priority is $prty$, and whose action is α' such that either $\alpha = \alpha'$, or α and α' are *independent*. A pair of actions α and α' are independent if executing them in either order results in the same system behavior [33]. For example, $z = x$ and $y = x$ are two independent actions, but $x = y + 1$ and $z = x$ are not independent. r is *nonshared* otherwise.

The goal of the Merge operator is to construct a model that contains shared behaviors of the input models as normal behaviors and nonshared behaviors as variabilities. To represent variabilities, we use parameterization [34]: Nonshared transitions are guarded by conditions denoting the transitions' origins before being lifted to the merge. Nonshared states can be lifted without any provisions—these states are reachable only via nonshared (and hence guarded) transitions.

Below, we describe our procedure for constructing a merge. We denote by $M_1 +_\rho M_2 = (S_+, \hat{s}_+, <_h^+, E_+, V_+, R_+)$ the merge of M_1 and M_2 with respect to ρ—a correspondence relation between these models.

1. **States and initial state.** (S_+ and $\hat{s}_+$) The set S_+ of states of $M_1 +_\rho M_2$ has one element for each tuple in ρ

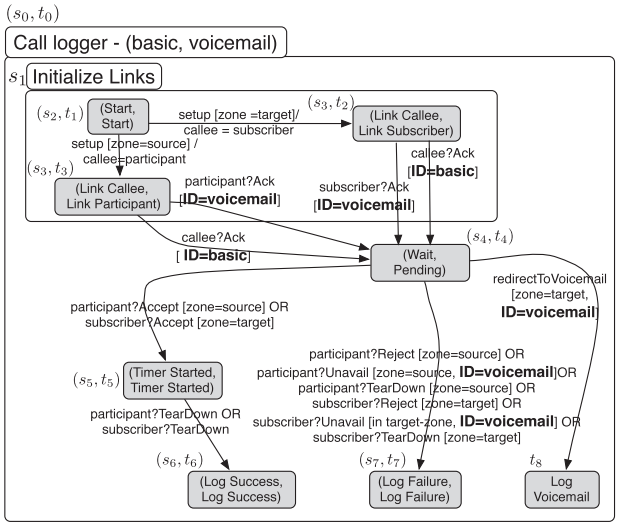

Fig. 10. Resulting merge for the call logger variants in Fig. 1.

and one element for each state in M_1 and M_2 that is nonshared. The initial state of $M_1 +_\rho M_2$, $\hat{s}_+$, is the tuple $(\hat{s}, \hat{t})$.

2. **Events and variables.** (E_+ and V_+) The set E_+ of events of $M_1 +_\rho M_2$ is the union of those of M_1 and M_2. The set V_+ of variables of $M_1 +_\rho M_2$ is the union of those of M_1 and M_2 plus a reserved enumerated variable **ID** that accepts values M_1 and M_2.
3. **Hierarchy tree.** ($<_h^+$) The hierarchy tree $<_h^+$ of $M_1 +_\rho M_2$ is computed as follows:

   a. **OR-states.** Let s be an OR-state in M_1 (the case for M_2 is symmetric), and let s' be a child of s.

      i. If s is mapped to t by ρ:

         A. If s' is mapped to a child t' of t by ρ, make (s', t') a child of (s, t) in $M_1 +_\rho M_2$ (see Fig. 11a).
         B. Otherwise, if s' is nonshared, make s' a child of (s, t) in $M_1 +_\rho M_2$ (see Fig. 11b).

      ii. Otherwise, if s is nonshared:

         A. If s' is mapped to a state t' by ρ, make (s', t') a child of s in $M_1 +_\rho M_2$ (see Fig. 11c).
         B. Otherwise, if s' is nonshared, make s' a child of s in $M_1 +_\rho M_2$ (see Fig. 11d).

   b. **AND-states.** Let s be an AND-state in M_1 (the case for M_2 is symmetric), and let $s_0, \ldots s_n$ be parallel substates of s.

      i. Let t be an AND-state in M_2, and let $t_0, \ldots t_m$ be parallel substates of t. If s is mapped to t by ρ, make (s, t) an AND-state in $M_1 +_\rho M_2$ and

         A. if s_i $(0 \leq i \leq n)$ is mapped to t_j $(0 \leq j \leq m)$ of t by ρ, make (s_i, t_j) a child of (s, t) in $M_1 +_\rho M_2$ (see Fig. 12a).
         B. otherwise, if s_i $(0 \leq i \leq n)$ is nonshared, make s_i a child of (s, t) in $M_1 +_\rho M_2$ (see Fig. 12b).

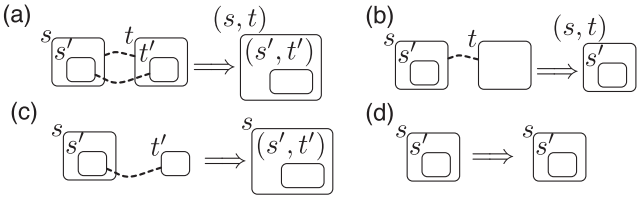

Fig. 11. Merging different OR-state patterns (note that part (d) refers to the case where both s and s' are nonshared).

      ii. Otherwise, if s is nonshared:

         A. If s_i $(0 \leq i \leq n)$ is mapped to a state t' by ρ, make (s_i, t') a child of s in $M_1 +_\rho M_2$ (see Fig. 12c).
         B. Otherwise, if s_i $(0 \leq i \leq n)$ is nonshared, make s_i a child of s in $M_1 +_\rho M_2$ (see Fig. 12d).

4. **Transition relation.** (R_+) The transition relation R_+ of $M_1 +_\rho M_2$ is computed as follows: Let $r = \langle s, a, c, \alpha, s', prty \rangle$ be a transition in M_1 (the case for M_2 is symmetric).

   i **(Shared transitions)** If r is shared, add to R_+ a transition corresponding to r with event a, condition c, action α (if $\alpha = \alpha'$) or action $\alpha; \alpha'$ (if $\alpha \neq \alpha'$), and priority $prty$.

   Note that according to the definition of shared transitions, α and α' are independent. Moreover, based on our assumptions in Section 3, M_1 and M_2 do not interact, i.e., α does not trigger any transition of M_2, and similarly, α' does not trigger any transition of M_2. Hence, the order of concatenation of α and α' in the merged model is not important. Moreover, in case $\alpha = \alpha'$, we keep only one copy of α in the merge. Hence, the merge does not execute the same action twice.

   ii (**Nonshared transitions**) Otherwise, if r is nonshared, add to R_+ a transition corresponding to r with event a, condition $c \wedge [\mathbf{ID} = M_1]$, action α, and priority $prty$.

As an example, Fig. 10 shows the resulting merge for the models of Fig. 1 with respect to the relation ρ in Fig. 8d. The conditions shown in boldface in Fig. 10 capture the origins of the respective transitions. For example, the transition from (s_4, t_4) to t_8 annotated with the condition **ID=voicemail** indicates a variable behavior that is applicable only for clients subscribing to voicemail.

Our definition of shared transitions is conservative in the sense that it requires such transitions to have identical events, conditions, and priorities in both input models. This is necessary in order to ensure that merges are behaviorally sound and do not introduce additional nondeterminism. However, such a conservative approach may result in redundant transitions which arise due to logical or unstated relationships between the events and conditions used in the input models. For example, in Fig. 10, the transitions from (s_2, t_1) to (s_3, t_2) and to (s_3, t_3) fire actions callee = subscriber and callee=participant, respectively. Thus, in state (s_3, t_3), the value of callee is equal to participant, and in state (s_3, t_2) to subscriber. This allows us to replace the event callee?Ack**[ID=basic]** on transition from (s_3, t_2) to (s_4, t_4)

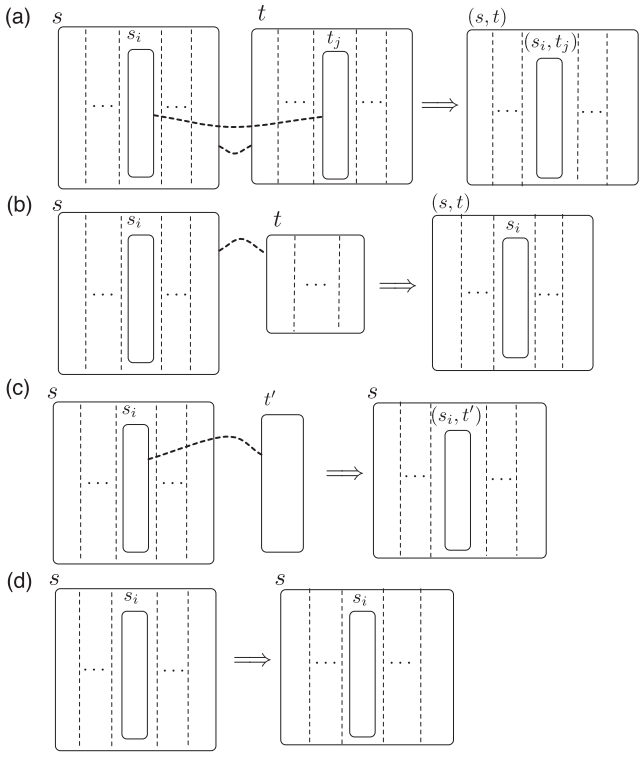

Fig. 12. Merging different AND-state patterns (note that part (d) refers to the case where both s and s' are nonshared).

by subscriber?Ack[**ID=basic**] and merge the two outgoing transitions from (s_3, t_2) into a single transition with label subscriber?Ack. Similarly, the two transitions from (s_3, t_3) to (s_4, t_4) can be merged into one transition with label participant?Ack. Identifying such redundancies and addressing them requires human intervention.

6 Tool Support

We have implemented our Match and Merge operators, respectively, described in Sections 4 and 5, as part of a tool called TReMer+. TReMer+ additionally provides implementations for the structural merge approach in [15] and the consistency checking approach in [35], [21], which we do not discuss here. TReMer+ consists of approximately 18,200 lines of Java code, of which 8,750 lines implement the graphical user interface, 8,450 lines implement the tool's core functions (model matching, model merging, traceability, and serialization), and 1,000 lines implement the glue code for interacting with an external consistency rule checker. The Merge operator described in this paper accounts for approximately 1,200 lines of the code—and the Match operator for approximately 2,000 lines, excluding the handling of the operators' input and output. The implementation of N-gram and linguistic matching [26], approximately 1,000 lines, is reused from existing open-source implementations. We have used TReMer+ for matching and merging the sets of variant Statechart models obtained from AT&T. Our tool and the material for the case studies conducted with it are available at http://se.cs.toronto.edu/index.php/TReMer+.

The main characteristic of TReMer+ is that it enables developers to make the relationships between models explicit and to treat these relationships as *first-class artifacts*

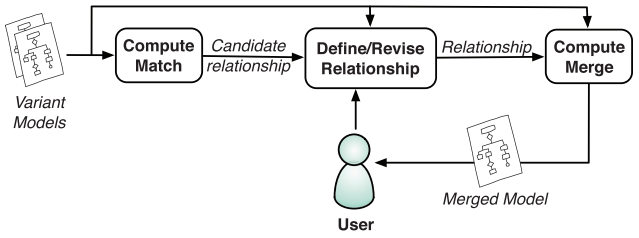

Fig. 13. Tool support overview.

[1]. Such treatment makes it possible to build alternative relationships between models—identified manually or based on results of our Match operator—and study the result of merge for each alternative.

Fig. 13 shows how our Match and Merge operators are applied in TReMer+: Given a pair of variant models, the user has a choice between defining a relationship manually or getting automated assistance from the Match operator. If the Match operator is applied, the user can still manually revise the computed relationship as she finds appropriate. The input models along with a (user-defined or user-adjusted) relationship are then used to compute a merge, presented to the user for further analysis. This may lead to the discovery of new element mappings or the invalidation of some of the existing ones. The user may then want to start a new iteration by revising the relationship and executing the subsequent activities.

In the remainder of this section, we illustrate our tool using the variant models in Fig. 1. First, the input models are specified using the tool's graphical editing environment. The user can then construct a relationship using the tool's mapping window, a snapshot of which is shown in Fig. 14. In this window, the input models are shown side-by-side. The user has the option to invoke the Match operator from the Tools menu to automatically compute a mapping between the states of the two models. This same window allows users to graphically specify or revise the state mappings. To establish a mapping, the user first clicks on a state of the model on the left and then on a state of the model on the right. To unmap a state, the user clicks on that state followed by a right click. To show the desired relationship, we have augmented the screenshot with a set of dashed lines indicating the related states. The relationship shown in the snapshot is the one given in Fig. 8d. Note that the tool represents hierarchical states using an arrow with a hollow tail from each substate to its immediate superstate. For example, the arrow from start to initialize_Link (right side of Fig. 14) indicates that initialize_Link is the immediate superstate of start. The merge computed by the tool with respect to the relationship defined above is shown in Fig. 15. As seen from the figure, nonshared behaviors are guarded by conditions denoting the input model exhibiting those behaviors.

7 Properties of Match

Our approach to matching is valuable if it offers a quick way to identify appropriate matches with reasonable accuracy, in particular in situations where matches are hard to find by hand, for example where the models are complex or the developers are less familiar with them. Here, we present some steps to evaluate our Match

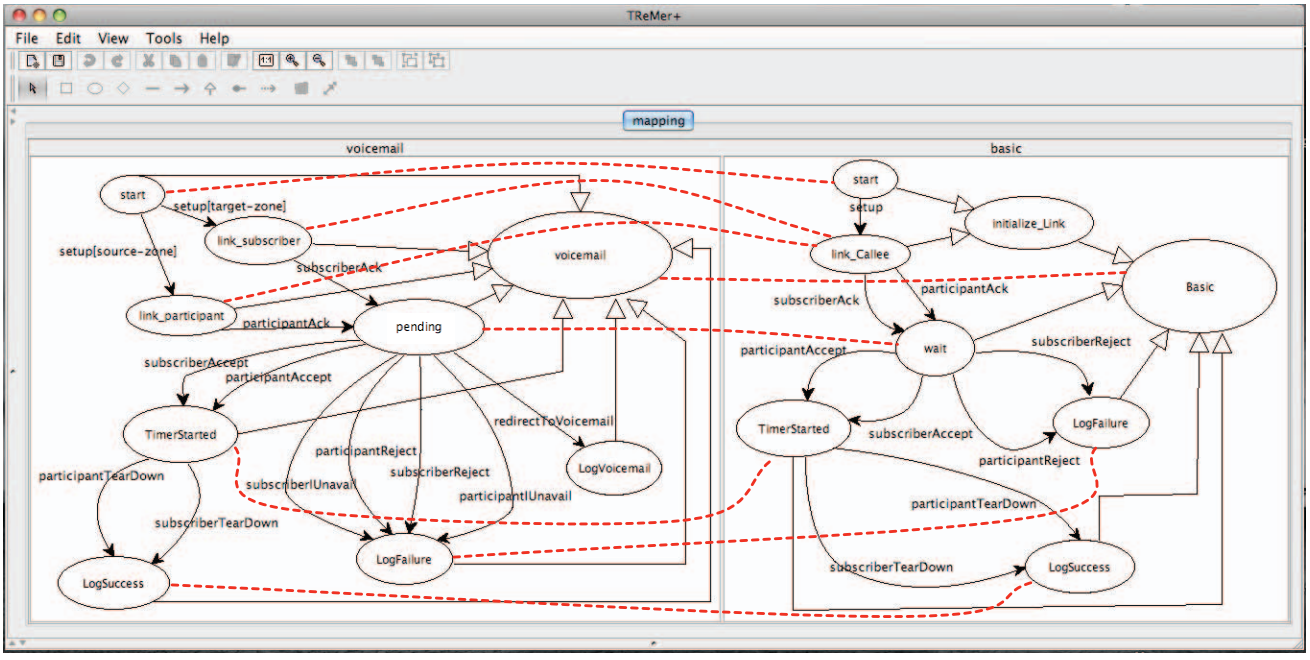

Fig. 14. Relationship between the models in Fig. 1.

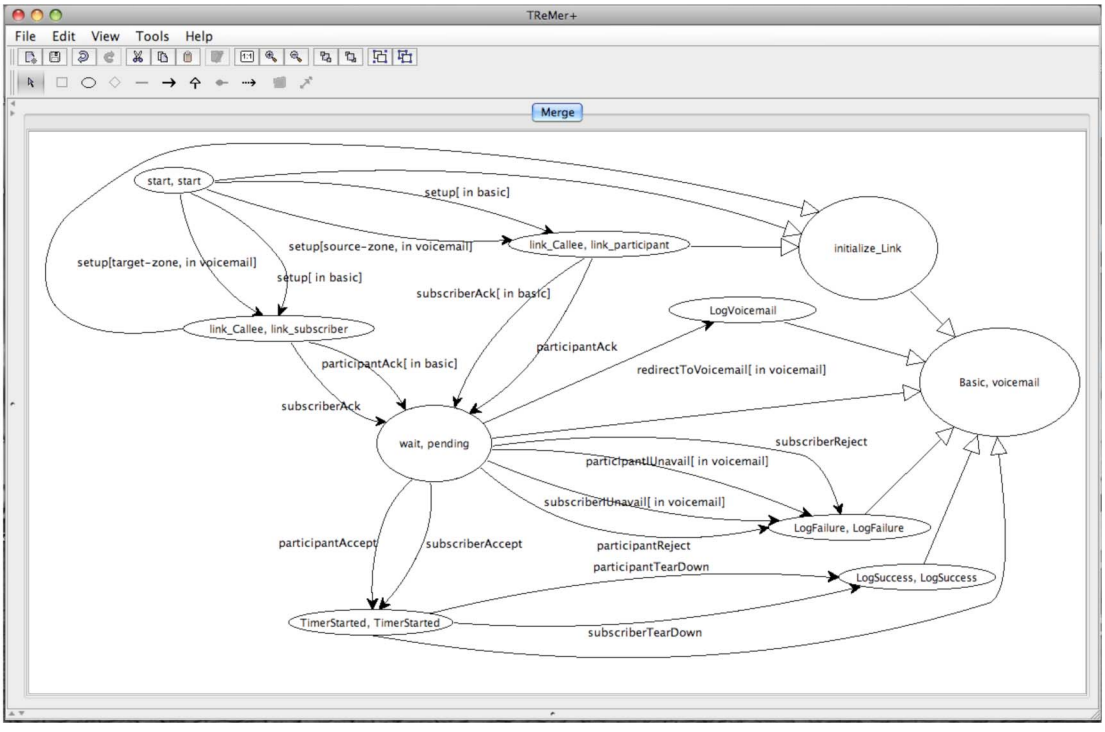

Fig. 15. Merge with respect to the relationship in Fig. 14.

operator. First, we discuss its complexity to show that it scales, and then we assess this operator by measuring the accuracy of the relationships it produces, when compared to the assessment of a human expert.

7.1 Complexity of Match

Let n_1 and n_2 be the number of states in the input models and let m_1 and m_2 be the number of transitions in these models. The space and time complexities of computing typographic and linguistic similarity scores between individual pairs of name labels are negligible and bounded by a constant, i.e., the largest value of similarity scores of the state names. Note that since the set of states names is finite and determined, we can compute such a bound. The space complexity of Match is then the storage needed for keeping a state similarity matrix and a label similarity matrix (L in Section 4.2) and is $O(n_1 \times n_2 + m_1 \times m_2)$. The time complexity of static matching is $O(n_1 \times n_2)$ and of behavioral matching $O(c \times m_1 \times m_2)$, where c is the maximum allowed number of iterations for the behavioral matching algorithm.

TABLE 1
Number of States and Transitions of the Studied Variant Models

Feature	Variant I		Variant II		All Correct Matches
	# states	# transitions	# states	# transitions	
Call Logger	18	40	21	63	11
Remote Identification	24	44	19	31	12
Parallel Location	28	71	33	68	16

7.2 Evaluation of Match

As with all heuristic matching techniques, the results of our Match operator should be reviewed and adjusted by users to obtain a desired correspondence relation. In this sense, a good way to evaluate a matcher is by considering the number of adjustments users need to make to the results it produces. A matcher is effective if it neither produces too many incorrect matches (false positives) nor misses too many correct matches (false negatives).

We use two well-known information retrieval metrics [36], namely, *precision*, and *recall*, to capture this intuition. Precision measures quality (i.e., a low number of false positives) and is the ratio of correct matches found to the total number of matches found. Recall measures coverage (i.e., a low number of false negatives) and is the ratio of the correct matches found to the total number of all correct matches. For example, if our matcher produces the relationship in Fig. 8b and the desired relation is as shown in Fig. 8d, the precision and recall are 8/14 (57 percent) and 8/8 (100 percent), respectively.

A good matching technique should produce high precision and high recall. However, these two metrics tend to be inversely related: Improvements in recall come at the cost of reducing precision and vice versa. In many circumstances, either precision or recall is more important than the other. For example, a web searcher would like every result on the first page to be relevant (high precision), but perhaps is not interested in retrieving all the relevant documents (recall can be low). In contrast, in most retrieval tasks for software engineering applications, software developers are willing to tolerate a small decrease in precision if it can bring about a comparable increase in recall [37]. We expect this to be true for model matching as well, especially for large models: It is easier for users to remove incorrect matches rather than find missing ones. For example, consider the desired relation in Fig. 8d: When our matcher produces the relation in Fig. 8b, its precision and recall rates are 0.57 and 1.0, respectively. While the precision may seem low, consider that our matcher already excluded 58 false matches from the 64 incorrect possibilities in Fig. 1. Of course, precision should not be too low—anything under 50 percent is an indication that more than half of the found matches are incorrect, and in the worst case, this means that the users require more effort to remove incorrect matches and find the missing ones than to do the entire matching manually!

We evaluated the precision and recall of our Match operator by applying it to a set of Statechart models describing different telecom features at AT&T. The fifth author of this paper acted as the domain expert for assessing correct matches. We studied three pairs of models, describing variant specifications of telecom features at AT&T. One of these is the call logger feature described in Section 1.1. Simplified versions of the variants of this feature were shown in Fig. 1. The other two features are *remote identification* and *parallel location*. Remote identification is used for authenticating a subscriber's incoming calls. Parallel location, also known as *find me*, places several calls to a subscriber at different addresses in an attempt to find her.

In Table 1, we show some characteristics of the studied models. For example, the first variant of the remote identification feature has 24 states and 44 transitions, and the second one has 19 states and 31 transitions. The correct relation (as identified manually by our domain expert) consists of 12 pairs of states. The Statechart models of these features are available in [38].

To compare the overall effectiveness of static matching, behavioral matching, and their combination, we computed their precision and recall for thresholds ranging from 0.95 down to 0.5. The results are shown in Fig. 16. As stated earlier in Section 4.3, threshold refers to the cutoff value used for determining the correspondence relation from the similarity degrees. The three diagrams at the top of Fig. 16 represent the precision values for the three case studies in

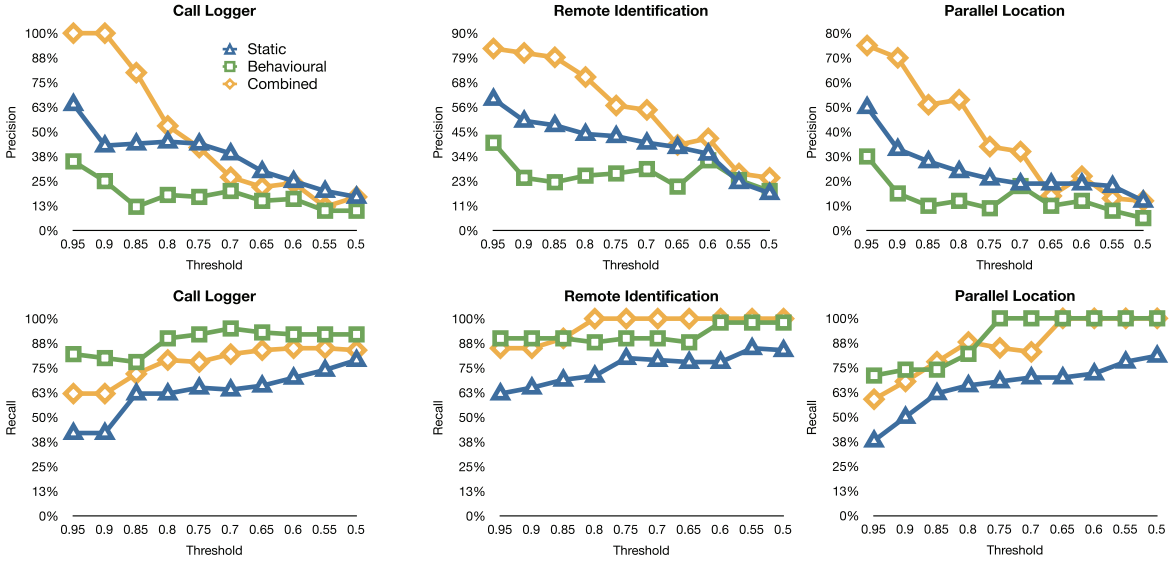

Fig. 16. Results of static, behavioral, and combined matching.

TABLE 2
Tradeoff Points between Precision and Recall Values for the Studied Variant Models

Feature	Threshold	Precision	Recall	F_2-Measure	Precision after Sanity Checks	F_2-Measure after Sanity Checks
Call Logger	0.9	100%	62%	0.71	100%	.71
	0.85	80%	72%	0.74	90%	.77
	0.8	54%	80%	0.68	77%	.78
Remote Identification	0.9	81%	85%	0.84	90%	.86
	0.85	79%	90%	0.86	85%	.88
	0.8	70%	100%	0.87	80%	.92
Parallel Location	0.9	70%	68%	0.69	85%	.73
	0.85	51%	78%	0.66	62%	.72
	0.8	53%	88%	0.72	65%	.79

Table 1, and the three diagrams at the bottom of Fig. 16 represent the recall values for those case studies. For example, when threshold is set to 0.9, the precision values for the combined, static, and behavioral matchings for the remote identification case study are 81, 50, and 24 percent, respectively. In our study we observed that among the three pairs of model variants, the parallel location variants had the largest overlap, i.e., the highest number of tuples in their correspondence relation, while the call logger variants had the least overlap. Furthermore, the hierarchy trees of the parallel location variants were the most similar in terms of the height of the hierarchy trees.

In the studied models, states with typographically similar names were likely to correspond. Hence, typographic matching and, by extension, static matching have high precision. However, static matching misses several correct matches, and hence has low recall. Behavioral matching, in contrast, has lower precision, but high recall. When the threshold is set reasonably high, combined matching has precision rates higher than those of static and behavioral matching on their own. This indicates that static and behavioral matching are filtering out each other's false positives. Recall remains high in the combined approach, as static matching and behavioral matching find many complimentary high-quality matches.

The results in Fig. 16 show that for each of the studied models, our combined matcher can achieve high precision (above 75 percent) for some thresholds and high recall (above 95 percent) for some other thresholds. To be able to compare the precision and recall values across different experiments, we use another metric known as *F-Measure* [36] which computes the harmonic mean of recall and precision and therefore is often used for comparative purposes. In this paper, we use a definition of F-Measure, known as F_2-Measure, which weighs recall values more highly than precision:

$$F_2\text{-Measure} = \frac{3 \times \text{Precision} \times \text{Recall}}{(2 \times \text{Precision}) + \text{Recall}}.$$

This weighting is appropriate in our domain where it is important to recall as many of the correct matches as possible.

Table 2 (columns 1 to 5 from left) presents recall, precision and threshold values that yield maximal F_2-Measure values across our three case studies. The maximal F_2-Measure values are obtained when threshold is set between 0.8 and 0.9. The precision values in the table range between 51 and 100 percent, and their corresponding recall values—between 62 and 100 percent.

As we anticipated, trying to improve our matcher's precision *and* recall by tweaking the heuristics behind it resulted in improving one at the expense of the other. For instance, when we let the behavioral heuristics run for relatively few iterations (<10), the result has higher precision but lower recall because more iterations are needed to properly propagate the similarity values in order to identify all of the correct matches. On the other hand, running the behavioral heuristics for a relatively large number of iterations (>100) causes a higher recall but a lower precision. Instead, we suggest the following strategy aimed at improving both precision and recall: 1) adjust the matcher's heuristics to optimize recall, and 2) identify and prune false positives using the sanity checks defined in Section 5.1. For example, applying these to the relation in Fig. 8b results in the removal of the tuples (s_1, t_0) and (s_1, t_1) (see Fig. 8c). This leads to an increase in precision from 8/14 (57 percent) to 8/12 (67 percent) and recall remains the same, i.e., 8/8 (100 percent). Fig. 17 shows the precision values of the combined matcher for the three studied models after applying the sanity checks. Note that the three curves in this figure correspond to the three case studies, rather than to the static/behavioral/combined algorithms as in Fig. 16. Compared to the results in Fig. 16, this methodology yields a 5-25 percent improvement in precision across the different thresholds. The last two columns in Table 2 represent the improved precision values after applying sanity checks and the corresponding F_2-Measures for thresholds 0.9, 0.85, and 0.8. Note that applying sanity checks does not change the recall values. It is possible that defining and applying additional sanity checks (e.g., by adding domain specificity) would improve precision even further.

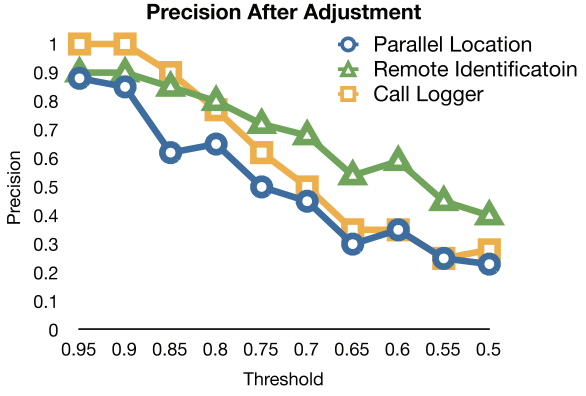

Fig. 17. Precision rates for the case study models in Table 1 after applying sanity checks in Section 5.1.

8 Properties of Merge

Our approach to merge is useful if it produces semantically correct results and scales well. Here, we discuss the complexity of our Merge operator and assess it by proving that it preserves the behavioral properties of the input models.

8.1 Complexity of Merge

The space complexity of Merge is linear in the size of the correspondence relation ρ and the input models. Theoretically, the size of ρ is $O(n_1 \times n_2)$. In practice, we expect the size of ρ to be closer to $\max(n_1, n_2)$, giving us linear space complexity for practical purposes. This was indeed the case for our models (see Table 1). The time complexity of Merge is $O(m_1 \times m_2)$.

8.2 Correctness of Merge

In this section, we prove that the merge procedure described in Section 5.2 is behavior preserving (see Appendix XI-C, available in the online supplemental material, for a detailed formal proof). The proof is based on showing that the merge is related to each of the input models via behavior preserving refinement relations. Basing the notion of behavioral merge on refinement relations is standard [12], [39]. Such relations capture the process of combining behaviors of individual models while preserving all of their agreements. Our notion of merge and our behavioral formalisms, however, have some notable differences with the existing work [12], [39], summarized below.

Nonclassical state machine formalisms have been previously defined to capture *partiality* [40]. Such models have two types of transitions: for describing definite and partial behaviors. In our work, in contrast, we use nonclassical state machines to explicitly capture behavioral variabilities. Variant models differ on some of their behaviors, i.e., those that are nonshared, giving rise to variabilities. We use parameterized Statecharts to explicitly differentiate between shared behaviors, i.e., those that are common between all variants, and nonshared behaviors, i.e., those that differ from one variant to another. Specifically, in these Statechart models, transitions labeled by a condition on the reserved variable **ID** represent the nonshared behaviors, and the rest of the transitions—the shared ones.

Definition 2 (Parameterized statecharts). *A parameterized Statechart M is a tuple $(S, \hat{s}, <_h, E, V, R^{shared}, R^{nonshared})$, where $M^{shared} = (S, \hat{s}, <_h, E, V, R^{shared})$ is a Statechart representing shared behaviors, i.e., containing the transitions not labeled with* **ID**, *and $M^{nonshared} = (S, \hat{s}, <_h, E, V, R^{nonshared})$ is a Statechart representing nonshared behaviors, i.e., containing only the transitions with* **ID**. *We denote the set of both shared and nonshared transitions of a parameterized Statechart by $R^{all} = R^{shared} \cup R^{nonshared}$, and we let $M^{all} = (S, \hat{s}, <_h, E, V, R^{all})$.*

When a partial model evolves and goes through refinement steps, the definite behaviors remain intact, but the partial behaviors may turn into definite or prohibited behaviors [40]. We define a new notion of refinement over parameterized Statechart models where 1) the nonshared behaviors are preserved through refinement, but the shared ones may turn into nonshared, and 2) the union of shared and nonshared behaviors does not change by refinement. That is, a model is more refined if it can capture more behavioral variabilities without changing the overall union of commonalities and variabilities. For example, the parameterized Statechart in Fig. 10 refines both models in Fig. 1 because it preserves all the behaviors of the models in Fig. 1, and further, it captures more behavioral variabilities. The models in Fig. 1 are parameterized Statecharts without nonshared behavior.

Since refinement relations are behavior preserving [31], [41], for parameterized Statechart models M_1 and M_2 where M_1 refines M_2 we have:

1. The set of behaviors of M_2 is a subset of the set of behaviors of M_1. That is, as we refine, we do not lose any behavior.
2. The set of shared behaviors of M_1 is a subset of the set of shared behaviors of M_2. That is, as we refine, we may increase behavioral differences.

Theorem 1. *Let M_1 and M_2 be (parameterized) Statechart models, let ρ be a correspondence relation between M_1 and M_2, and let $M_1 +_\rho M_2$ be their merge as constructed in Section 5.2. Then, $M_1 +_\rho M_2$ refines both M_1 and M_2.*

The above theorem proves that our merge procedure in Section 5.2 generates a *common refinement* of M_1 and M_2. The complete proof of this theorem is given in Appendix XI-C, available in the online supplemental material. Given this theorem and the property-preservation result of refinement relation mentioned above, we have:

1. Behaviors of the individual input models, M_1 and M_2, are present as either shared, i.e., unguarded, or nonshared, i.e., guarded, behaviors in their merge, $M_1 +_\rho M_2$. Thus, the merge preserves all positive traces of the input models. For example, the positive behaviors $\mathbf{P_1}$ and $\mathbf{P_2}$ in Fig. 2 are both preserved in the merge in Fig. 10: $\mathbf{P_1}$ as an unguarded behavior and $\mathbf{P_2}$ an a guarded behavior.
2. The set of shared, i.e., unguarded, behaviors of $M_1 +_\rho M_2$ is a subset of the behaviors of the individual input models, M_1 and M_2. Therefore, any behavior absent from either input is absent from the unguarded fragment of their merge. In other words, any negative behavior, i.e., safety property, that holds over the input models also holds over the unguarded fragment of their merge.
3. The guarded (nonshared) behaviors of the input models M_1 and M_2 are preserved in $M_1 +_\rho M_2$, i.e., merge preserves behavioral disagreements. But the unguarded (shared) behaviors of M_1 and M_2 may become nonshared in $M_1 +_\rho M_2$, i.e., merge can turn behavioral agreements into disagreements. For example, the transition t_4 to t_7 in Fig. 1 represents an unguarded (shared) behavior of the voicemail variant. But it turns into a guarded (nonshared) behavior in the merge, as exemplified by the transition from (s_4, t_4) to t_8 in Fig. 10.

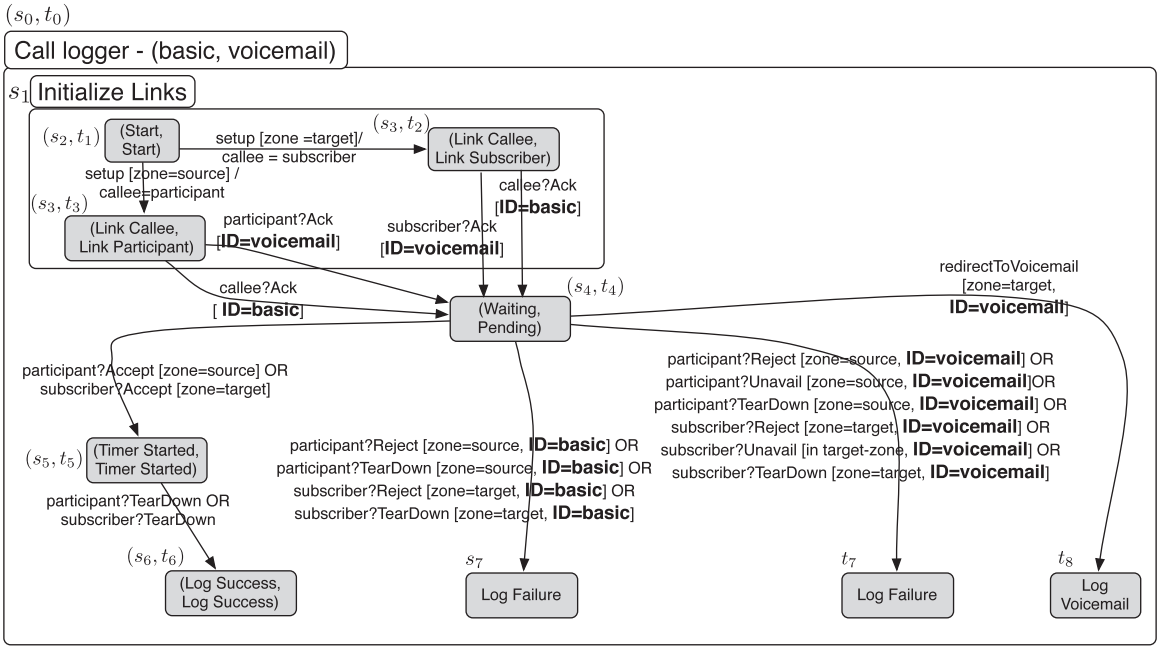

Fig. 18. The merge of the models in Fig. 1 with respect to the relation in Fig. 8d when (s_7, t_7) is removed from the relation.

In short, the merge includes, in either guarded or unguarded form, *every* behavior of the input models. The use of parameterization for representing behavioral variabilities allows us to generate behavior-preserving merges for models that may even be inconsistent.

A change in the correspondence relation (ρ) does not cause any behaviors to be added to or removed from the merge, but may make some guarded behaviors unguarded, or vice versa. For example, if we remove the tuple (s_7, t_7) from the correspondence relation ρ in Fig. 8d, the resulting merge is the model in Fig. 18. The model in this figure still preserves every behavior of the input models, but has more parameterized behaviors, e.g., the transitions from (s_4, t_4) to s_7 and t_7.

Our merge construction respects transition priorities, thus ensuring that no new nondeterminism is introduced. Section 5 described our procedure for merging *pairs* of models. It can be extended to n-ary merges by iteratively merging a new input model with the result of a previous merge, except the reserved variable **ID** (in the merge procedure of Section 5.2) will need to range over subsets of the input model indices. In this case, the order in which the binary merges are applied does not affect the final result.

9 Discussion

In this section, we compare our approach to related work, and discuss the results presented in this paper and the practical considerations of some of our decisions.

9.1 Structural versus Behavioral Merge

Approaches to model merging can be categorized into two main groups based on the mathematical machinery that they use to specify and automate the merge process [42]: 1) approaches based on algebraic graph-based techniques, and 2) approaches based on behavior preserving relations. Approaches in the first group view models as graphs and formalize the relationships between models using graph homomorphisms that map models directly or indirectly through connector models [15], [43]. These approaches, while being general, are not particularly suitable for merging behavioral models because model relationships are restricted to graph homomorphisms, which are tools for preserving model *structure* rather than *behavioral* properties.

We show the difference between structure-preserving and behavior-preserving merges using a simple example. Consider the models M_1 and M_2 in Figs. 19a and 19b, and let $\rho = \{(s_0, t_0), (s_1, t_1), (s_1, t_2), (s_2, t_3), (s_2, t_4)\}$. The model in Fig. 19c shows a merge of M_1 and M_2 that preserves the structure of the input models: It is possible to embed each of M_1 and M_2 into M_{1+2} using graph homomorphisms. This merge, however, does not preserve the behaviors of M_1 and M_2 because it collapses two behaviorally distinct states t_1 and t_2 into a single state r_1 in the merge. The model in Fig. 19d is an alternative merge of M_1 and M_2 which is constructed based on the notion of state machine refinement as proposed in this current paper: It can be shown that M'_{1+2} refines both M_1 and M_2. As shown in the figure, states t_1 and t_2 are, respectively, lifted to two distinct states, q_1 and q_2, in this merge. By basing merge on refinement, we can choose to keep states in the merged model distinct even if ρ maps them to one single state in the other model. The flexibility to duplicate states of the source models in the merge is essential for behavior preservation but is not supported by the merge approaches that are based on graph homomorphisms.

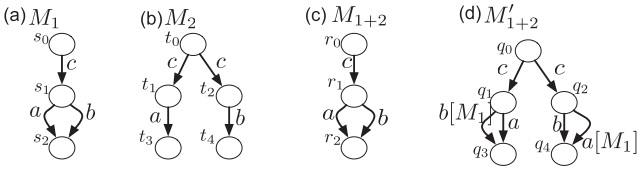

Fig. 19. (a) Model M_1, (b) model M_2, (c) M_{1+2}: a possible merge of M_1 and M_2 that preserves their behaviors, and (d) M'_{1+2}: a possible merge of M_1 and M_2 that preserves their structure.

9.2 Merging Models with Behavioral Discrepancies

Approaches to behavioral model merging generally specify merge as a common behavioral refinement of the original models. However, these approaches differ on how they handle discrepancies between models both in their vocabulary and their behaviors. Larsen et al. [44] show that behavioral common refinements can be logically characterized as the conjunction of the original specifications when models are consistent and have the same vocabulary. Fischbein et al. [13] introduce a notion of *alphabet refinement* that allows merging models with different vocabulary but consistent behaviors. The main focus of [13] is to use merge as a way to elaborate partial models with unspecified vocabulary or unknown, but consistent, behaviors. Huth and Pradhan [45] merge partial behavioral specifications where a dominance ordering over models is given to resolve their potential inconsistencies. These approaches do not provide support for merging models with behavioral variabilities such as those presented in Fig. 1.

9.3 Analytical Reasoning for Matching Transition Labels

We have explored the use of analytical reasoning for comparing transition labels. The N-gram algorithm, which is used in this paper for computing matching values for transition labels, is not suitable for comparing complex mathematical expressions. For example, it would find a rather small degree of similarity between mathematical expressions $(x \wedge y) \vee z$ and $(x \vee z) \wedge (y \vee z)$, whereas analytical reasoning, e.g., by a theorem prover, would identify these expressions as identical. While we did not encounter the need for such analysis on our case study, it might be necessary for such domains as web services, where transition labels may include complex program fragments.

9.4 Overlapping, Interacting and Cross-Cutting Behavioral Models

Models which have been developed in a distributed manner may relate to one another in a variety of ways. The nature of the relationships between such models depends primarily on the intended application of the models and how they were developed [46]. The work presented in this paper focuses on merging a collection of interrelated models when relationships describe *overlaps* between the models' behaviors. Alternatively, relationships may describe shared interfaces for *interaction*, in particular when models are independent components or features of a system, or the relationships may describe ways in which models *alter* one another's behavior (e.g., a cross-cutting model applied to other models) [47]. The former kind of relationships is studied in the model *composition* literature where the goal is to assemble a set of autonomous but interacting features that run sequentially or in parallel (e.g., [33], [48], [17], [31], [11]). Unlike merge, composition is concerned with how models communicate with one another through their interfaces rather than how they overlap in content. The latter relationships are studied in the area of aspect-oriented development where the goal is to *weave* cross-cutting concerns into a base system (e.g., [49], [50], [51]). The focus of this paper was on relationships that capture overlaps between model behaviors, and not on situations where model relationships describe interactions or cross-cutting aspects.

9.5 Model Matching Techniques

Approaches to model matching can be *exact* or *approximate*. Exact matching is concerned with finding structural or behavioral conformance relations between models. Graph homomorphisms are examples of the former, whereas simulation and bisimulation relationships are the latter. Finding exact correspondences between models has applications in many fields, including graph rewriting, pattern recognition, program analysis, and compiler optimization. However, it is not very useful for matching distributed models because the vocabularies and behaviors of these models seldom fit together in an exact way, and thus, exact conformance relations between these models are unlikely to be found.

Most domains use heuristic techniques for matching. These techniques yield values denoting a likelihood of correspondence between elements of different models. In database design, finding correspondences between database schemata is referred to as schema matching [52]. State-of-the-art schema matchers, such as Protoplasm [9], combine several heuristics for computing similarities between schema elements. Our typographic and linguistic heuristics (Section 4.1) are very similar to those used in schema matching, but our other heuristics are tailored to behavioral models.

Several approaches to matching have been proposed in software engineering. Maiden and Sutcliffe [6] employ heuristic reasoning for finding analogies between a problem description and already existing domain abstractions. Ryan and Mathews [53] use approximate graph matching for finding overlaps between concept graphs. Alspaugh et al. [54] propose term matching based on project glossaries for finding similarities between textual scenarios. Mandelin et al. [4] combine diagrammatic and syntactic heuristics for finding matches between architecture models. Xing and Stroulia [55] use heuristic-based name-similarity and structure-similarity matchers to identify conceptually similar entities in UML class diagrams. None of these approaches were specifically designed for behavioral models and are either inapplicable or unsuitable for matching Statechart models.

Some matching approaches deal with behavioral models of a different kind. For example, Kelter and Schmidt [56] discuss differencing mechanisms specifically designed for UML Statecharts. This work assumes that models are developed centrally within a unified modeling environment. Other approaches apply to independently developed models. Lohmann [57] and Zisman et al. [8] define similarity measures between web-services to identify candidate services to replace a service in use when it becomes unavailable or unsuitable due to a change. Quante and Koschke [58] propose similarity measures between finite state automata generated by different reverse engineering mechanisms to compare the effectiveness of these mechanisms. Bogdanov and Walkinshaw [59] provide an algorithm for comparing LTSs without relying on the initial state or any particular states of the underlying models as the reference point. None of these are applicable to both our particular purpose (comparing different versions of the same feature) and to our particular class of models (Statecharts models). Further, the evaluation of our matching technique is targeted at behavioral models built in the telecommunication domain. To our knowledge, the usefulness and effectiveness of model matching have not been studied in this context before.

In computer science theory, several notions of behavioral conformance have been proposed to capture the behavioral similarity between models with quantitative features such as time or probability [60]. For these models, a discrete notion of similarity, i.e., models are either equivalent or they are not, is not helpful because minor changes in the quantitative data may cause equivalent models to become inequivalent, even if the difference between their behaviors is very minor. Therefore, instead of equivalences that result in a binary answer, one needs to use relations that can differentiate between slightly different and completely different models. Examples of such relations are stochastic or Markovian notions of behavioral similarity (e.g., [61], [62]). Our formulation of behavioral similarity (Section 4.2) is analogous to these similarity relations. The goal of this work is to define a distance metric over the space of (quantitative) reactive processes and study the mathematical properties of the metric. Our goal, instead, is to obtain a similarity measure that can detect pairs of states with a high degree of behavioral similarity.

9.6 Model Merging Techniques

Model merging spans several application areas. In database design, merge is an important step for producing a schema capturing the data requirements of all the stakeholders [2]. Software engineering deals extensively with model merging—several papers study the subject in specific domains, including early requirements [15], static UML diagrams [63], [14], [64], [65], [66], and declarative specifications [67]. None of these were specifically designed for behavioral models and are either inapplicable or unsuitable for matching Statechart models. We compared our work with existing approaches for merging behavioral models in Section 9.2.

There has also been work on defining languages for model merging, e.g., the Epsilon Merging Language (EML) [10] and Atlas Model Management Architecture (AMMA) [68]. EML is a rule-based language for merging models with the same or different metamodels. The language distinguishes four phases in the merge process and provides flexible constructs for defining the rules that should be applied in each phase. AMMA facilitates modeling tasks such as merging using model transformation languages defined between different metamodels. Despite their versatility, the current versions of EML and AMMA do not formalize the conditions and consequences of applying the merge rules, and hence, in contrast to our approach, do not provide a formal characterization of the merge operation when applied to behavioral models.

In this paper, we focused on the application of behavioral merge as a way to reconcile models developed independently. Behavioral merge operation may arise in several other related areas, including program integration [69] and merging declarative specifications [67]. These approaches share the same general motivation with our work, which is preservation of semantics and support for handling inconsistencies. However, they are not targeted at consolidating variant specifications and, further, do not use Statecharts as the underlying notation.

Several approaches to variability modeling have been proposed in software maintenance and product line engineering. For example, Halmans and Pohl [70] provide an elaborate view of modeling variability in use cases by distinguishing between aspects essential for satisfying customers' needs and those related to the technical realization of variability. Our merge operator makes use of parameterization for representing variabilities between different models. This is a common technique in modeling behavioral variability in Statechart models [34]. A similar parameterization technique has been used in [71] for capturing variability in Software Cost Reduction (SCR) tables [72].

In requirements and early design model merging, discrepancies are often treated as inconsistencies [73], [12], [15]. Some of these approaches require that only consistent models be merged [12]. Others tolerate inconsistency and can represent the inconsistencies explicitly in the resulting merged model [73], [15]. Our work is similar to the latter group as we explicitly model variabilities between models using parameterization.

Several approaches provide guidelines and methodologies for building product lines out of legacy systems (e.g., [74]). Most of these approaches rely on a manual review of code, design, and documentation of the system that can be time consuming. The ability to mine legacy product lines and automate their translation to a product line model capturing commonalities and variabilities is a necessity. Clearly, such translations should preserve the set and behavior of existing products, and, potentially, allow identification and addition of new products to the product line. Our approach can provide a basis for behavior-preserving refactoring of product line models from a set of existing (legacy) model variants [75].

9.7 Handling Additional Statechart Features

Our approach to Match and Merge can be systematically extended to handle Statechart models with features other than those discussed in the paper. The general strategy for implementing such extensions is 1) to modify Definition 1 to incorporate the new features in the Statechart base notation, 2) to identify the additional sanity conditions (to be added to those defined in Section 5.1) so that the new features in the original models are properly lifted to the merge, and 3) to modify the Merge algorithm in Section 5.2 to deal with shared aspects of the input models so that the merge remains semantically sound. For example, one possible solution is to constrain the model relationships via additional sanity checks so that the new features always fall in the nonshared parts of the input models. Although this solution works for arbitrary features, it is too conservative. A better way is to study the features one by one to identify less constraining sanity conditions for each. Below, we include two examples:

- **State entry/exit/in/during actions.** For input models with state actions, we constrain the relationships so that they can only be one-to-one, i.e., a state in one model cannot be mapped to more than one state in the other model. This ensures that only one copy of every state action in the input models is lifted to the merge, and hence, these actions are executed the same number of times in the merge as in the input models. As observed in a study reported in [76], it is very common for models to be related via one-to-one relationships, so this is not a limiting restriction.

- **Internal events.** In this paper, we assumed that the input models do not include internal events. That is, a transition triggered by an external event cannot produce internal events that may, in turn, trigger other transitions (see Section 3). Our approach can handle internal events if the following two conditions hold:
  - The sanity checks are extended to ensure that the relationships between states in the original models are one-to-one. Hence, sequences of internal events are executed the same number of times in the merge as in the input models.
  - The definition of shared transitions in the merge algorithm in Section 5.2 is changed so that a pair of shared transitions has identical actions, i.e., $\alpha = \alpha'$, as well as identical events, conditions, and priority. These shared transitions are replaced in the merged model with one transition with an action α.

  Since the action on the transition in the merge is the same as in the corresponding transitions in the input models, the sequences of internal events generated in the merge by α are the same as in each of the input models.

Optionally, when extending the approach with additional Statechart features, we can also augment the existing Match heuristics with new heuristics designed specifically for these new features. For example, we can choose to compute similarity values based on the state entry/exit/in/during actions and use these values to initialize the behavioral matching in Section 4.2, i.e., we can consider these values to be the behavioral matching similarities at iteration zero (this approach is implemented in [75]).

9.8 Practical Limitations

Our work has a number of limitations which we list below.

Evaluation. Our evaluation in Section 7 may not be a comprehensive assessment of the effectiveness of our Match operator: First, in our evaluation we assume that it is possible to find a matching relationship which is agreeable to all users. In practice, this may not be the case [3]. A more comprehensive evaluation would require several independent subjects to provide their desired correspondence relations and use these for computing an average precision and recall. Second, matching results can be improved by proper user guidance, which we did not measure here. More specifically, in Section 7, we evaluated Match as a fully automatic operator. In practice, it might be reasonable to use Match interactively, with the user seeding it with some of the more obvious relations, and pruning incorrect relations iteratively. We expect that such an approach will improve accuracy. Alternatively, a developer might prefer to assess the output of the Match operator by computing merge and inspecting the resulting model for validity. This way, each correspondence relation is treated as a hypothesis for how the models should be combined, to be adjusted if the resulting merge does not make sense. We plan to investigate the feasibility of this approach further.

Scalability and usability. Discussions in Sections 7.1 and 8.1 show that the computational complexity of our operators is not high. Since the space complexity of merge is linear in the size of the input models, the size of the merge does not grow as rapidly as the size of (parallel) compositions [33], and hence, issues such as the state-explosion problem [33] do not arise in our work. In our evaluation and experimentation with the Match and Merge operators, the actual running times of our algorithms were also negligible.

Although our matching and merging algorithms scale well in terms of computational efficiency and space usage, there are some issues regarding usability of our approach that may limit its applicability to large models. In particular, currently TReMer+ cannot preserve the layout of the source models, and ignores all their visual cues during merge. We plan to address this issue in the future. Another 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 such 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.

10 Conclusions and Future Work

In this paper, we presented an approach to matching and merging of Statechart models. Our Match operator includes heuristics that use both static and behavioral properties to match pairs of states in the input models. Our evaluations show that this combination produces higher precision than relying on static or behavioral properties alone. Our Merge operator produces a combined model in which variant behaviors of the input models are parameterized using guards on their transitions. The result is a merge that preserves the temporal properties of the input models. We have also developed a tool that implements both our Merge and Match operators and enables seamless application of the two.

While our evaluation results demonstrate the effectiveness of our approach, its practical utility can only be assessed by more extensive empirical studies. The value of our tool is likely to depend on factors such as the size and complexity of the models, the user's familiarity with the models, and the user's subjective judgment of the matching results. Our Merge operator only applies to hierarchical state machine models. Extending it to behavioral models described in different notations, i.e., heterogeneous behavioral models, presents a challenge. In future work, we plan to address this limitation by developing ways to merge models at a logical level.

The work reported in this paper is part of a larger ongoing project on model management and its applications to software engineering. A vision for this project has been presented in [1]. Our current direction is to develop appropriate model management operators for the suite of UML notations and to provide a unifying framework for using these operators in a cohesive way [77], [78]. Developing such a framework involves a careful analysis of a wide range of concerns such as assumptions about the nature of models and their intended use, the details of the relationships between models (e.g., level of granularity, semantics, representation), and the correctness criteria expected from the model management operators. We have already developed a preliminary classification of these concerns for model management operators [79] and intend to expand on and refine this classification in the future.

The ideas behind our work have already been picked up for different purposes in different areas, most notably, in Product line engineering for characterizing different notions of feature composition [80] and for developing behavior-preserving methods to build product line models from products [75], in web-service discovery for identifying candidate services to repair or modify service compositions [57], and, recently, in computational cognitive modeling for managing and composing cognitive models with complex relationships [81].

ACKNOWLEDGMENTS

The authors are grateful to Thomas Smith and Gregory Bond for their help with their analysis of telecom features and the ECharts language. They thank the *TSE* reviewers and members of the Model Management Research Group at the University of Toronto for their insightful comments. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada, IBM, and the Research Council of Norway under the ModelFusion project.

REFERENCES

[1] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh, "A Manifesto for Model Merging," *Proc. Workshop Global Integrated Model Management Colocated with ICSE '06,* 2006.

[2] P. Bernstein, "Applying Model Management to Classical Meta Data Problems," *Proc. First Biennial Conf. Innovative Data Systems Research,* pp. 209-220, 2003.

[3] S. Melnik, *Generic Model Management: Concepts and Algorithms.* Springer, 2004.

[4] D. Mandelin, D. Kimelman, and D. Yellin, "A Bayesian Approach to Diagram Matching with Application to Architectural Models," *Proc. 28th Int'l Conf. Software Eng.,* pp. 222-231, 2006.

[5] G. Spanoudakis and A. Finkelstein, "Reconciling Requirements: A Method for Managing Interference, Inconsistency and Conflict," *Annals of Software Eng.,* vol. 3, pp. 433-457, 1997.

[6] N. Maiden and A. Sutcliffe, "Exploiting Reusable Specifications through Analogy," *Comm. ACM,* vol. 35, no. 4, pp. 55-64, 1992.

[7] J. Rubin and M. Chechik, "A Declarative Approach for Model Composition," *Proc. Workshop Model-Driven Approaches in Software Product Line Eng. Colocated with SPLC '10,* 2010.

[8] A. Zisman, G. Spanoudakis, and J. Dooley, "A Framework for Dynamic Service Discovery," *Proc. IEEE/ACM 23rd Int'l Conf. Automated Software Eng.,* pp. 158-167, 2008.

[9] P. Bernstein, S. Melnik, M. Petropoulos, and C. Quix, "Industrial-Strength Schema Matching," *SIGMOD Record,* vol. 33, no. 4, pp. 38-43, 2004.

[10] D. Kolovos, R. Paige, and F. Polack, "Merging Models with the Epsilon Merging Language (EML)," *Proc. Ninth Int'l Conf. Model Driven Eng. Languages and Systems,* pp. 215-229, 2006.

[11] S. Nejati, M. Sabetzadeh, M. Chechik, S. Uchitel, and P. Zave, "Towards Compositional Synthesis of Evolving Systems," *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 285-296, 2008.

[12] S. Uchitel and M. Chechik, "Merging Partial Behavioural Models," *Proc. 12th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 43-52, 2004.

[13] D. Fischbein, G. Brunet, N. D'Ippolito, M. Chechik, and S. Uchitel, "Weak Alphabet Merging of Partial Behaviour Models," *ACM Trans. Software Eng. and Methodology,* to Appear, 2010.

[14] A. Mehra, J.C. Grundy, and J.G. Hosking, "A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design," *Proc. IEEE/ACM 20th Int'l Conf. Automated Software Eng.,* pp. 204-213, 2005.

[15] M. Sabetzadeh and S. Easterbrook, "View Merging in the Presence of Incompleteness and Inconsistency," *Requirements Eng. J.,* vol. 11, no. 3, pp. 174-193, 2006.

[16] J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios," *Proc. 22nd Int'l Conf. Software Eng.,* pp. 314-323, May 2000.

[17] M. Jackson and P. Zave, "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services," *IEEE Trans. Software Eng.,* vol. 24, no. 10, pp. 831-847, Oct. 1998.

[18] P. Zave, "Modularity in Distributed Feature Composition," *Software Requirements and Design: The Work of Michael Jackson,* B. Nuseibeh and P. Zave, eds., Good Friends Publishing, 2010.

[19] G. Bond, E. Cheung, H. Goguen, K. Hanson, D. Henderson, G. Karam, K. Purdy, T. Smith, and P. Zave, "Experience with Component-Based Development of a Telecommunication Service," *Proc. Eighth Int'l Symp. Component-Based Software Eng.,* pp. 298-305, 2005.

[20] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave, "Matching and Merging of Statecharts Specifications," *Proc. 29th Int'l Conf. Software Eng.,* pp. 54-64, 2007.

[21] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik, "Global Consistency Checking of Distributed Models with TReMer+," *Proc. 30th Int'l Conf. Software Eng.,* pp. 815-818, 2008.

[22] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The Statemate Approach.* McGraw Hill, 1998.

[23] J. Niu, J.M. Atlee, and N.A. Day, "Template Semantics for Model-Based Notations," *IEEE Trans. Software Eng.,* vol. 29, no. 10, pp. 866-882, Oct. 2003.

[24] G. Bond, "An Introduction to ECharts: The Concise User Manual," AT&T, technical report, http://echarts.org/Downloads/Download-document/An-Introduction-to-ECharts-The-Concise-User-Manual-2008-05-20-v1.3-beta.html, 2008.

[25] G. Bond and H. Goguen, "ECharts: Balancing Design and Implementation," AT&T technical report, http://echarts.org, 2002.

[26] C. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing.* MIT Press, 1999.

[27] T. Pedersen, S. Patwardhan, and J. Michelizzi, "WordNet: Similarity-Measuring the Relatedness of Concepts," *Proc. 19th Nat'l Conf. Assoc. for the Advancement of Artificial Intelligence,* pp. 1024-1025, 2004,

[28] S. Patwardhan and T. Pedersen, "Using WordNet-Based Context Vectors to Estimate the Semantic Relatedness of Concepts," *Proc. Workshop Making Sense of Sense—Bringing Computational Linguistics and Psycholinguistics Together,* pp. 1-8, 2006.

[29] P. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining.* Addison Wesley, 2005.

[30] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms,* second ed. The MIT Press, 2001.

[31] R. Milner, *Communication and Concurrency.* Prentice-Hall, 1989.

[32] R.D. Nicola, U. Montanari, and F. Vaandrager, "Back and Forth Bisimulations," *Proc. Eighth Int'l Conf. Concurrency Theory,* pp. 152-165, 1990.

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

[34] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures,* first ed. Addison Wesley, 2004.

[35] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik, "Consistency Checking of Conceptual Models via Model Merging," *Proc. IEEE 15th Int'l Requirements Eng. Conf.,* pp. 221-230, 2007.

[36] M. McGill and G. Salton, *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[37] J.H. Hayes, A. Dekhtyar, and J. Osborne, "Improving Requirements Tracing via Information Retrieval," *Proc. IEEE 11th Int'l Symp. Requirements Eng.,* pp. 138-147, 2003.

[38] S. Nejati, "Behavioural Model Fusion," PhD dissertation, Univ. of Toronto, 2008.

[39] A. Hussain and M. Huth, "On Model Checking Multiple Hybrid Views," *Proc. First Int'l Symp. Leveraging Applications of Formal Methods,* pp. 235-242, 2004.

[40] K. Larsen and B. Thomsen, "A Modal Process Logic," *Proc. Third Ann. Symp. Logic in Computer Science,* pp. 203-210, 1988.

[41] M. Huth, R. Jagadeesan, and D.A. Schmidt, "Modal Transition Systems: A Foundation for Three-Valued Program Analysis," *Proc. 10th European Symp. Programming,* pp. 155-169, 2001.

[42] M. Sabetzadeh, "Merging and Consistency Checking of Distributed Models," PhD dissertation, Univ. of Toronto, 2008.

[43] H. Liang, Z. Diskin, J. Dingel, and E. Posse, "A General Approach for Scenario Integration," *Proc. 11th Int'l Conf. Model Driven Eng. Languages and Systems,* pp. 204-218, 2008.

[44] K. Larsen, B. Steffen, and C. Weise, "A Constraint Oriented Proof Methodology Based on Modal Transition Systems," *Proc. First Int'l Workshop Tools and Algorithms for Construction and Analysis of Systems,* pp. 17-40, 1995.

[45] M. Huth and S. Pradhan, "Model-Checking View-Based Partial Specifications," *Electronic Notes Theoretical Computer Science,* vol. 45, pp. 1-23, 2001.

[46] S. Nejati and M. Chechik, "Behavioural Model Fusion: An Overview of Challenges," *Proc. ICSE Workshop Modeling in Software Eng.,* 2008.

[47] B. Nuseibeh, J. Kramer, and A. Finkelstein, "A Framework for Expressing the Relationships between Multiple Views in Requirements Specification," *IEEE Trans. Software Eng.,* vol. 20, no. 10, pp. 760-773, Oct. 1994.

[48] J. Hay and J. Atlee, "Composing Features and Resolving Interactions," *Proc. Eighth ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 110-119, 2000.

[49] A. Moreira, A. Rashid, and J. Araújo, "Multi-Dimensional Separation of Concerns in Requirements Engineering," *Proc. IEEE 10th Int'l Symp. Requirements Eng.,* pp. 285-296, 2005.

[50] P. Tarr, H. Ossher, W. Harrison, and S.S. Jr, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proc. 21st Int'l Conf. Software Eng.,* pp. 107-119, 1999.

[51] W. Harrison, H. Ossher, and P. Tarr, "General Composition of Software Artifacts," *Proc. Fifth Int'l Symp. Software Composition, Colocated with ETAPS '06,* pp. 194-210, 2006.

[52] E. Rahm and P. Bernstein, "A Survey of Approaches to Automatic Schema Matching," *The VLDB J.,* vol. 10, no. 4, pp. 334-350, 2001.

[53] K. Ryan and B. Mathews, "Matching Conceptual Graphs as an Aid to requirements Re-Use," *Proc. IEEE Int'l Symp. Requirements Eng.,* pp. 112-120, 1993.

[54] T. Alspaugh, A. Antón, T. Barnes, and B. Mott, "An Integrated Scenario Management Strategy," *Proc. IEEE Fourth Int'l Symp. Requirements Eng.,* pp. 142-149, 1999.

[55] Z. Xing and E. Stroulia, "Differencing Logical UML Models," *Automated Software Eng.,* vol. 14, no. 2, pp. 215-259, 2007.

[56] U. Kelter and M. Schmidt, "Comparing State Machines," *Proc. Workshop Comparison and Versioning of Software Models,* pp. 1-6, 2008.

[57] N. Lohmann, "Correcting Deadlocking Service Choreographies Using a Simulation-Based Graph Edit Distance," *Proc. Sixth Int'l Conf. Business Process Management,* pp. 132-147, 2008.

[58] J. Quante and R. Koschke, "Dynamic Protocol Recovery," *Proc. 14th Working Conf. Reverse Eng.,* pp. 219-228, 2007.

[59] K. Bogdanov and N. Walkinshaw, "Computing the Structural Difference between State-Based Models," *Proc. 16th Working Conf. Reverse Eng.,* pp. 177-186, 2009.

[60] F. van Breugel, *Proc. Workshop Approximate Behavioural Equivalences,* 2008.

[61] L. de Alfaro, M. Faella, and M. Stoelinga, "Linear and Branching Metrics for Quantitative Transition Systems," *Proc. 31st Int'l Colloquium Automata, Languages and Programming,* pp. 97-109, 2004.

[62] O. Sokolsky, S. Kannan, and I. Lee, "Simulation-Based Graph Similarity," *Proc. 12th Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems,* pp. 426-440, 2006.

[63] M. Alanen and I. Porres, "Difference and Union of Models," *Proc. Sixth Int'l Conf. Unified Modeling Language,* pp. 2-17, 2003.

[64] K. Letkeman, "Comparing and Merging Uml Models in IBM Rational Software Architect," IBM, technical report, http://www-306.ibm.com/software/awdtools/architect/swarchitect/, 2006.

[65] A. Zito, Z. Diskin, and J. Dingel, "Package Merge in UML 2: Practice vs. Theory?" *Proc. Ninth Int'l Conf. Model Driven Eng. Languages and Systems,* pp. 185-199, 2006.

[66] A. Boronat, J. Carsí, I. Ramos, and P. Letelier, "Formal Model Merging Applied to Class Diagram Integration," *Electronic Notes Theoretical Computer Science,* vol. 166, pp. 5-26, 2007.

[67] D. Jackson, "Alloy: A Lightweight Object Modelling Notation," *ACM Trans. Software Eng. and Methodology,* vol. 11, no. 2, pp. 256-290, 2002.

[68] M.D.D. Fabro, J. Bézivin, F. Jouault, and P. Valduriez, "Applying Generic Model Management to Data Mapping," *Proc. Journées Bases de Données Advancées,* 2005.

[69] S. Horwitz, J. Prins, and T. Reps, "Integrating Noninterfering Versions of Programs," *ACM Trans. Programming Languages and Systems,* vol. 11, no. 3, pp. 345-387, 1989.

[70] G. Halmans and K. Pohl, "Communicating the Variability of a Software-Product Family to Customers," *Software and System Modeling,* vol. 2, no. 1, pp. 15-36, 2003.

[71] S. Faulk, "Product-Line Requirements Specification (PRS): An Approach and Case Study," *Proc. IEEE Fifth Int'l Symp. Requirements Eng.,* pp. 48-55, 2001.

[72] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw, "SCR*: A Toolset for Specifying and Analyzing Requirements," *Proc. Ann. Conf. Computer Assurance,* pp. 109-122, 1995.

[73] S. Easterbrook and M. Chechik, "A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints," *Proc. 23rd Int'l Conf. Software Eng.,* pp. 411-420, 2001.

[74] J. Bayer, J. Girard, M. Würthner, J. DeBaud, and M. Apel, "Transitioning Legacy Assets to a Product Line Architecture," *Proc. Seventh European Software Eng. Conf. Held Jointly with the Seventh ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 446-463, 1999.

[75] J. Rubin and M. Chechik, "Quality of Behavior-Preserving Product Line Refactorings," 2011.

[76] J. Whittle, A. Moreira, J. Araújo, P. Jayaraman, A. Elkhodary, and R. Rabbi, "An Expressive Aspect Composition Language for Uml State Diagrams," *Proc. 10th Int'l Conf. Model Driven Eng. Languages and Systems,* pp. 514-528, 2007.

[77] R. Salay, M. Chechik, S. Easterbrook, Z. Diskin, P. McCormick, S. Nejati, M. Sabetzadeh, and P. Viriyakattiyaporn, "An Eclipse-Based Tool Framework for Software Model Management," *Proc. OOPSLA Workshop Eclipse Technology eXchange,* pp. 55-59, 2007.

[78] R. Salay, "Using Modeler Intent in Software Engineering," PhD dissertation, Univ. of Toronto, 2010.

[79] M. Chechik, S. Nejati, and M. Sabetzadeh, "A Relationship-Based Approach to Model Integration," *Innovations in Systems and Software Eng.,* To Appear, 2011.

[80] A. Classen, P. Heymans, T.T. Tun, and B. Nuseibeh, "Towards Safer Composition," *Proc. 31st Int'l Conf' Software Eng. Companion,* pp. 227-230, 2009.

[81] "Researching and Developing Persistent and Generative Cognitive Models," http://palm.mindmodeling.org/pgcm/Welcome.html, 2010.

[82] R. Alur, S. Kannan, and M. Yannakakis, "Communicating Hierarchical State Machines," *Proc. 26th Int'l Colloquium Automata, Languages and Programming,* pp. 169-178, 1999.

[83] D. Dams, R. Gerth, and O. Grumberg, "Abstract Interpretation of Reactive Systems," *ACM Trans. Programming Languages and Systems,* vol. 2, no. 19, pp. 253-291, 1997.

Shiva Nejati received the MSc and PhD degrees from the University of Toronto in 2003 and 2008, respectively. She is currently a research scientist at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg. From 2009 to 2012, she was a researcher at the Simula Research Laboratory, Norway. Her main research area is software engineering, with specific interests in model-based development, behavior analysis, requirements engineering, specification and design methods, and web services. She is a member of the IEEE Computer Society.

Mehrdad Sabetzadeh received the PhD degree from the University of Toronto in 2008. He is currently a research scientist at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg. From 2009 to 2012, he was a member of the research staff at the Simula Research Laboratory, Norway. In 2009, he was a visiting researcher at University College London. His main research interests are focused on model-based software development with an particular emphasis on rmodel-based verification and validation of business- and mission-critical applications. He is a member of the IEEE Computer Society.

Marsha Chechik received the PhD degree from the University of Maryland in 1996. She is currently a professor and vice chair in the Department of Computer Science at the University of Toronto. Her research interests are in the application of formal methods to improve the quality of software. She has authored more than 90 papers in formal methods, software specification and verification, computer security, and requirements engineering. In 2002-2003, She was a visiting scientist at Lucent Technologies in Murray Hill, New York, and at Imperial College London. She is an associate editor of the *IEEE Transactions on Software Engineering* (2003-2007, 2010-present). She is a member of IFIP WG 2.9 on Requirements Engineering and an associate editor of the *Journal on Software and Systems Modeling*. She regularly serves on program committees of international conferences in the areas of software engineering and automated verification. She was a cochair of the 2008 International Conference on Concurrency Theory (CONCUR), program committee cochair of the 2008 International Conference on Computer Science and Software Engineering (CASCON), and program committee cochair of the 2009 International Conference on Formal Aspects of Software Engineering (FASE). She is a member of the IEEE Computer Society.

Steve Easterbrook received the PhD degree in 1991 in computing from Imperial College London. He is a professor of computer science at the University of Toronto and was a lecturer at the School of Cognitive and Computing Science, University of Sussex, from 1990 to 1995. In 1995, he moved to the US to lead the research team at NASA's Independent Verification and Validation (IV&V) Facility in West Virginia, where he investigated software verification on the Space Shuttle Flight Software, the International Space Station, the Earth Observation System, and several planetary probes. He moved to the University of Toronto in 1999. His research interests range from modeling and analysis of complex software systems to the socio-cognitive aspects of team interaction, including communication, coordination, and shared understanding in large software teams. He has served on the program committees for many conferences and workshops in requirements engineering and software engineering, and was general chair for RE '01 and program chair for ASE '06. In the summer of 2008, he was a visiting scientist at the UK Met Office Hadley Centre. He is a member of the IEEE Computer Society.

Pamela Zave received the AB degree in English from Cornell University and the PhD degree in computer sciences from the University of Wisconsin–Madison. She has held position at the University of Maryland and Bell Laboratories and is now with AT&T–Research. Her work on the foundations of requirements engineering has been recognized with three Ten-Year Most Influential Paper awards. Her research on telecommunication services has led to three Best Paper Awards, the AT&T Strategic Patent Award, the AT&T Science and Technology Medal, and 22 patents. She leads a group that has developed two successful large-scale IP-based telecommunication systems, and created an advanced tool suite that is being used by several current projects. She is also active in the areas of Internet architecture and languages for software modeling. She is chair of IFIP Working Group 2.3 on Programming Methodology, a member of the IEEE Computer Society, an AT&T fellow, and an ACM fellow.