

Matching and Merging of Statecharts Specifications

Shiva Nejati[†] Mehrdad Sabetzadeh[†] Marsha Chechik[†] Steve Easterbrook[†] Pamela Zave[‡]

[†]Department of Computer Science
University of Toronto, Toronto, ON, Canada
{shiva,mehrdad,chechik,sme}@cs.toronto.edu

[‡]AT&T Laboratories—Research
Florham Park, NJ, USA
pamela@research.att.com

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 hierarchical Statecharts: Match, for finding correspondences between models, and Merge, for combining models with respect to known correspondences between them. Our Match operator is heuristic, making use of both static and behavioural 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 behaviour through parameterization. In this way, we automatically construct merges that preserve the semantics of Statecharts models. We illustrate and evaluate our work by applying our operators to AT&T telecommunication features.

1 Introduction

Model-based development involves construction, integration, and maintenance of complex models. For large-scale projects, this can include many inter-related models, representing different versions over time, different variants across a product family, different options for implementation, and so on. *Model Management* aims to provide a systematic way to represent the relationships between models, and a set of operators for manipulating them. Such operators include *Match*, for finding correspondences between models, *Diff*, for finding differences between models, *Merge*, for putting together a set of models with respect to known relationships between them, and *Slice*, for producing a projection of a model based on a given criterion [2, 18, 6].

Among these operators, *Match* and *Merge* play a central role in supporting distribution and coordination of modelling tasks. In any situation where models are developed independently, *Match* provides a way to discover the relationships between them, for example, to compare variants [15], to identify inconsistencies [30], and to support

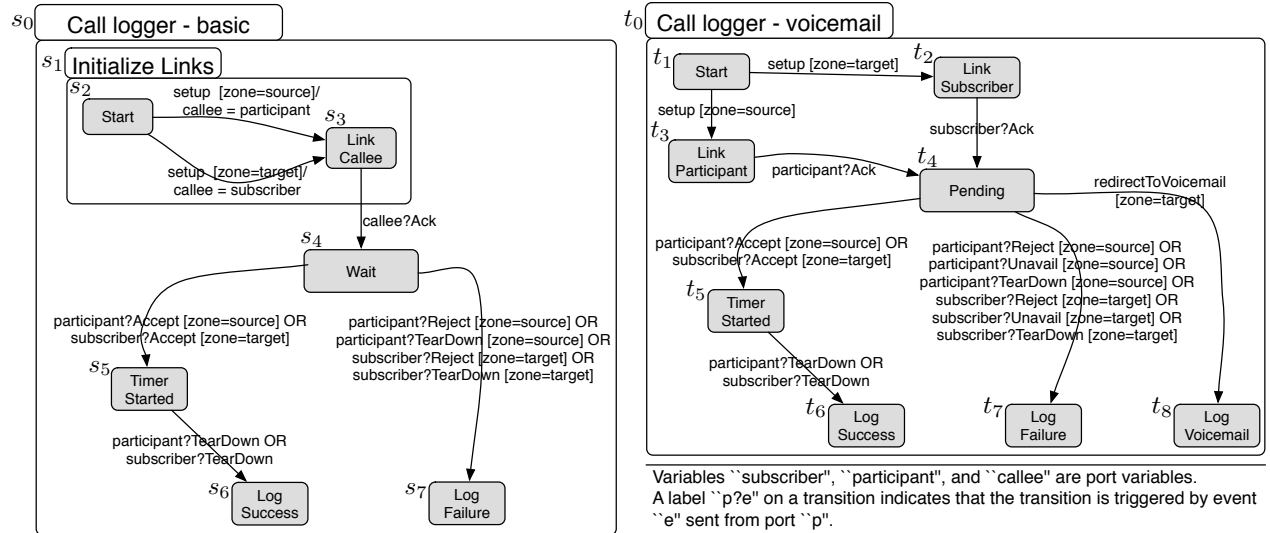
reuse [14]. Sophisticated *Match* tools, e.g. [3], can handle models that use different vocabularies and different levels of abstraction. *Merge* provides a way to combine models, to gain a unified perspective, to understand interactions between models, and to perform various types of analysis, e.g., validation and verification.

Many existing approaches to *Match* and *Merge* focus on structural similarities between models. For example, [18] studies matching and merging of conceptual database schemata; [17] proposes a general framework for merging visual design diagrams; [27] describes an algebraic approach for merging requirements views; and, [15] provides 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 modelling notations, and is particularly suited to early stages of development, when models may have loose or undefined semantics.

In contrast, recent work on behavioural models has concentrated on establishing semantic relationships between models. For example, [32] uses logical pre/post-conditions over object interactions for merging independently-developed sequence diagrams, and [31] uses refinement relations for merging consistent state-machine models such that their behavioural properties are preserved.

In this paper, we present an approach to matching and merging hierarchical Statecharts¹ models that exploits both structural and semantic information in the models, and ensures that behavioural properties are preserved. 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 behavioural properties are guaranteed to hold in their merge. We illustrate and evaluate our work by applying our operators to a set of AT&T telecommunication features.

¹Statecharts is a design and implementation language and is widely used for specifying dynamic behaviours of software systems [9].



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

Figure 1. Simplified variants of the call logger feature.

1.1 Motivating Example

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 [11], and form part of a consumer voice-over-IP service. In the current implementation of DFC [5], the features are written using 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., caller hanging up or callee not responding), a connection is not established, a failure is logged; otherwise, when the call is completed, information about the call is logged.

Initially, the functionality was designed only for basic phone calls, for which logging is limited to the direction of a call, the address location where a call is answered, success or failure, and the duration if it succeeds. Later, a variant of this feature 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. Figure 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 disjunction (OR).

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, re-

sulting in the construction of an independent model for each variant. To reduce the high costs associated with verifying and maintaining independent models, it is desirable to consolidate the variants of each feature into a single coherent model. To do this, we need to be able to identify correspondences between variant models and merge these models with respect to their correspondences.

1.2 Contributions of this Paper

Match and Merge are recurring problems arising in different contexts. Our motivating example illustrates one of the many applications of these operators. Implementation of 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 non-shared 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 the Statecharts notation. The contributions of this paper are as follows:

- A versatile Match operator for Statecharts (Section 4). 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 behavioural 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 Statecharts models (Section 6).

- A Merge operator for Statecharts (Section 5). We provide a procedure for constructing behaviour-preserving merges that also respect the hierarchical structuring of the input models.

2 Overview of Our Approach

The main challenge in devising a usable Match operator is finding a set of effective heuristics that can imitate the reasoning of a domain expert. In our work, we use two types of heuristics: static and behavioural. Static heuristics use semantic-free attributes, such as element names, for measuring similarities. For the models in Figure 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 behaviours. Our behavioural heuristic can find these pairs.

Our Match operator, produces a correspondence relation between states in the two models. For the models of Figure 1, it may yield the correspondence relation shown in Figure 3(b). 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 Figure 3(c).

In contrast to matching, merging is not heuristic, and is almost entirely automatable. Given a pair of models and a correspondence relation between them, our Merge operator automatically produces a merge that: (1) preserves the behavioural properties of the input models, (2) respects the hierarchical structure of these models, and (3) distinguishes between shared and non-shared behaviours of these models by attaching appropriate guard conditions to non-shared transitions. Figure 4, shows the merge of the models of Figure 1 with respect to the relation in Figure 3(c). In the merge, non-shared transitions are guarded by boldface conditions representing the models they originate from.

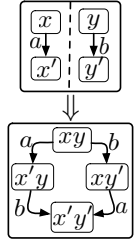
This merge is behaviour-preserving. For example, the property “After a connection is set up, a successful call will be logged if the subscriber or the participant sends Accept” holds in both models in Figure 1, and is thus preserved in their merge as a shared behaviour (denoted by the path from state (s_4, t_4) to (s_6, t_6)). The property “After a connection is set up, a voicemail will be logged if the call is redirected to the voicemail service”, which holds over the voicemail variant but not over the basic, is represented as a parameterized behaviour in the merge (denoted by the transition from (s_4, t_4) to t_8), and is preserved only when its guard holds. The merge also respects the hierarchical structure of the input models, providing users with a merge that has the same conceptual structure as the input models.

3 Background

While our work is general and can be applied to various Statecharts dialects, in this paper, we ground our discussion on a particular dialect, called ECharts [4]. ECharts provides well-defined deterministic semantics for the Statecharts language, and is suitable for detailed design and implementation. The AT&T telecom features are specified in ECharts.

A Statecharts model is a tuple $(S, \hat{s}, \preceq, E, V, R)$, where S is a finite set of states; $\hat{s} \in S$ is an initial state; \preceq is a partial order 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 $\langle s, a, c, \alpha, s', prty \rangle$, where $s, s' \in S$ are the transition’s source and target, respectively, $a \in E$ is the triggering event, c is an optional predicate over V , α is a sequence of zero or more actions, and $prty$ is a number denoting the transition’s priority. Each state in S can be either an atomic state or a superstate. The hierarchy tree \preceq defines a partial order on states with the top superstate as root and the atomic states as leaves. For example, in Figure 1, s_0 is the root, s_2 through s_7 are leaves, and s_1 is neither.

This formalism, adapted from [22], supports superstates (OR states), but not parallel states (AND states). ECharts uses parallel states with interleaved transition executions [4], and can be translated to the above formalism using the interleaving semantics of [22]. A simple example of this translation is shown on the right. In ECharts, transitions with the same event and condition can be made deterministic by assigning globally-ordered priorities to them (using $prty$). The models shown in Figure 1 are already deterministic, and thus, no prioritization is required.



We present our Match and Merge operators for input models $M_1 = (S_1, \hat{s}_1, \preceq_1, E_1, V_1, R_1)$ and $M_2 = (S_2, \hat{s}_2, \preceq_2, E_2, V_2, R_2)$. We assume the sets of events, E_1 and E_2 , and variables, V_1 and V_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.

4 Matching Statecharts

Our Match operator (Figure 2) uses a hybrid approach combining static matching (Section 4.1) and behavioural matching (Section 4.2). Static matching is independent of Statecharts semantics and uses typographic and linguistic similarities between state names, and similarities between state depths in the models’ hierarchy trees. Behavioural matching generates similarity degrees between states based on their behavioural semantics. We aggregate these static

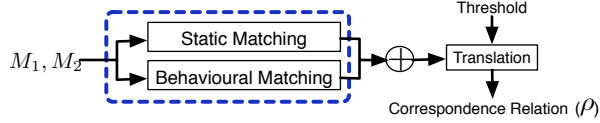


Figure 2. Overview of the Match operator.

and behavioural heuristics to produce 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 4.4).

4.1 Static Matching

Typographic Matching assigns to every pair (s, t) of states a normalized value in $[0..1]$ computed by applying the N-gram algorithm [16] 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., $N = 3$).

Linguistic Matching 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 [23] for this purpose.

Depth Matching uses state depths to derive a useful 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. Depth matching assigns a normalized value in $[0..1]$ to every pair (s, t) of states. The closer the depths of s and t in their respective hierarchy trees are, the closer this value is to one. Depth matching is not used when the input models are at different levels of abstraction.

4.2 Behavioural Matching

Our behavioural matching technique is reminiscent of deciding bisimilarity between state-machines [19]. Bisimilarity provides a natural way to characterize behavioural equivalence. Bisimilarity is a recursive notion and can be defined in two ways, forward and backward [21]. Two states are *forward bisimilar* if they can transition to (forward) bisimilar states via identically-labelled transitions; and are (forward) dissimilar otherwise. Backward bisimilarity is dual.

Bisimilarity relates states with *precisely* the same set of behaviours, but it cannot capture *partial* similarities. For example, states s_4 and t_4 in Figure 1 transit to (forward) bisimilar states s_7 and t_7 , respectively, with transitions labelled `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 behaviours differ on a few other transitions, e.g., the one labelled `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 behaviours 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 neighbours of s and those of t . By neighbours, we mean either successor/child states (forward neighbours) or predecessor/parent states (backward neighbours) 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 neighbour of s . That is, s has a transition to s' labelled a , or s' is child of s where a is a special label called *child*. Treating children states as neighbours allows us to propagate similarities from children to their parents.

Behavioural matching is a total function $\mathcal{B} : S_1 \times S_2 \rightarrow [0..1]$. We denote by $\mathcal{B}^i(s, t)$ the degree of similarity between states s and t after the i th 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 provides a mechanism for users to apply their domain expertise during the matching process. Since behavioural 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 behavioural 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. We improve transition label comparison by using the variable assignments in the action parts of transition labels for term rewriting. For example, in Figure 1, the actions `callee = participant` and `callee = subscriber` suggest that the transition label `callee?Ack` is similar to `participant?Ack` and `subscriber?Ack`. We account for this by (automatic) term replacement prior to applying the N-gram algorithm. The algorithm assigns a similarity value $L(a, b)$ in $[0..1]$ to every pair (a, b) of transition labels.

We have also explored the use of analytical reasoning for comparing transition labels. For example, the N-gram algorithm would find a rather small degree of similarity between conditions $(x \wedge y) \vee z$ and $(x \vee z) \wedge (y \vee z)$, whereas an-

alytical reasoning, e.g. by a theorem prover, would identify these conditions as identical. Our experimentation with this idea indicates that such reasoning over labels is expensive and also unnecessary because examples such as the above are not very common in practice.

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 neighbours 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 neighbours.

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 neighbours 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 neighbours of s and their best matches among the forward neighbours of t after iteration $i - 1$ 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 neighbours of t and their best matches among the forward neighbours 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 taking 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 neighbours of s and t , respectively. The larger the $\mathcal{B}^i(s, t)$, the more the behaviours of s and t are alike.

This 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 maximum number of iterations.

4.3 Combining Different Similarity Measures

To obtain 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. Learning-based techniques have been shown to be effective when proper training data is available [15]. At this stage, we do not have sufficient training data to employ such techniques. In our current implementation, we use a simple approach based on linear averages.

We generate an aggregate value for static heuristics, denoted by \mathcal{S} , by taking the maximum of typographic and linguistic similarities, and computing its weighted average

	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.

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

(b) A correspondence relation ρ . and sanity checks of Sec 5.1.

Figure 3. Results of matching for call logger.

with depth similarity. Behavioural similarity, \mathcal{B} , is computed as the maximum of forward behavioural and backward behavioural matching. To produce an overall combined measure, denoted \mathcal{C} , we take a weighted average of \mathcal{B} with \mathcal{S} . Figure 3(a) illustrates \mathcal{C} for the models in Figure 1. Here, we use a 4-to-1 ratio for averaging name similarities (max. of typographic and linguistic) with depth similarity, and use equal weights for averaging \mathcal{S} and \mathcal{B} . These weights, which we arrived at by experimentation, are also used for the evaluation in Section 6.

4.4 Translating Similarities to Correspondences

To obtain a correspondence relation between M_1 and M_2 , the user sets a threshold for translating the overall similarity degrees into a 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%, we obtain the correspondence relation ρ shown in Figure 3(b). Since matching is a heuristic process, ρ should be reviewed and, if necessary, adjusted by the user. We assume that the user would remove the spurious pairs (s_6, t_7) , (s_6, t_8) , (s_7, t_6) and (s_7, t_8) from ρ . As we will discuss in Section 5.1, the resulting relation needs to be further revised before merge.

5 Merging Statecharts

In this section, we describe our Merge operator for Statecharts. The input to this operator is a pair of models, M_1 and M_2 , and a correspondence relation ρ . The output is a merged model if ρ satisfies certain sanity checks. Otherwise, a subset of ρ violating the checks is identified.

5.1 Sanity Checks for Correspondence Relations

Before applying the Merge operator, we need to ensure that ρ passes certain sanity checks. To have behaviourally sound merges, the initial states of the input models should correspond. 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 behavioural properties of the models with the old initial states to those with the new initial states. For example, instead of evaluating a temporal property p at \hat{s} (resp. \hat{t}), we check AXp at \hat{s}' (resp. \hat{t}'), where AX denotes the universal next-time operator.

To construct merges that are structurally sound, ρ must satisfy the following conditions for every $(s, t) \in \rho$:

1. (*monotonicity*) If ρ relates a proper descendant of s (resp. t) to a state x in M_2 (resp. M_1), then x must be a proper descendant of t (resp. s).
2. (*relational adequacy*) 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 .

Monotonicity ensures that ρ does not relate an ancestor of s to t (resp. 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, \preceq .

Pairs in ρ that violate any of the above conditions are reported to the user. In our example, the relation shown in Figure 3(b) 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. We assume that the user addresses these conflicts by eliminating (s_1, t_0) and (s_1, t_1) from ρ . The resulting relation, shown in Figure 3(c), passes all sanity checks and can be used for merge.

5.2 Merge Construction

To merge M_1 and M_2 , we first need to identify their shared and non-shared parts with respect to ρ . A state x is *shared* if it is related to some state by ρ , and is *non-shared* 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 , and whose priority is $prty$. r is *non-shared* otherwise. Notice that there is no requirements for the actions of r and r' to be identical.

The goal of the Merge operator is to construct a model that contains shared behaviours of the input models as normal behaviours and non-shared behaviours as variabilities. To represent variabilities, we use parameterization [8]: Non-shared transitions are guarded by conditions denoting the transitions' origins, before being lifted to the merge. Non-shared states can be lifted without any provisions – these states are reachable only via non-shared (and hence, guarded) transitions.

Below, we describe our procedure for constructing a merge. We denote by $M_1 +_\rho M_2 =$

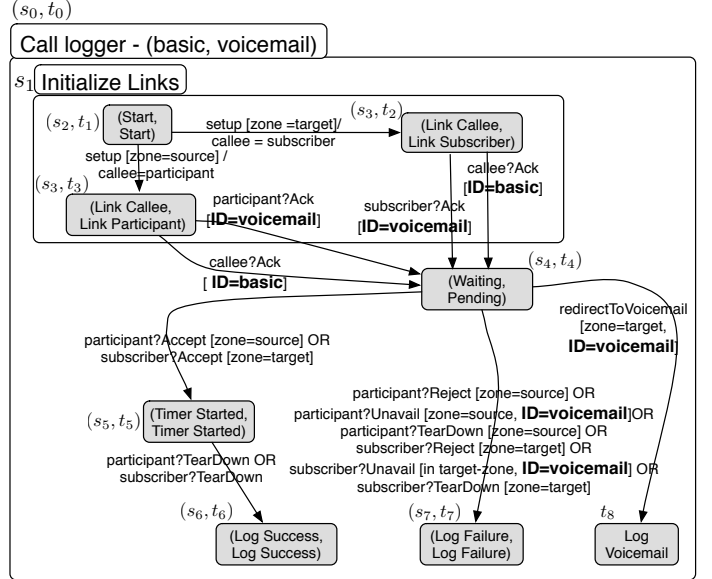


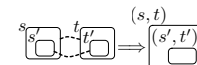
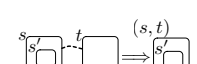
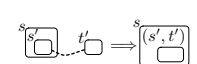
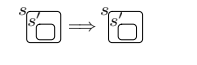
Figure 4. Resulting merge for call logger.

$(S_+, \hat{s}_+, \preceq_+, E_+, V_+, R_+)$ the merge of M_1 and M_2 with respect to ρ .

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

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 .

Hierarchy Tree. (\preceq_+) The hierarchy tree \preceq_+ of $M_1 +_\rho M_2$ is computed as follows. Let s be a superstate in M_1 (the case for M_2 is symmetric), and let s' be a child of s .

- if s is mapped to t by ρ ,
 - if s' is mapped to a child t' of t by ρ , make (s', t') a child of (s, t) in $M_1 +_\rho M_2$. 
 - otherwise, if s' is non-shared, make s' a child of (s, t) in $M_1 +_\rho M_2$. 
- otherwise, if s is non-shared
 - if s' is mapped to a state t' by ρ , make (s', t') a child of s in $M_1 +_\rho M_2$. 
 - otherwise, if s' is non-shared, make s' a child of s in $M_1 +_\rho M_2$. 

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).

- (**Shared Transitions**) if r is shared, add to R_+ a transition corresponding to r with event a , condition c , action² α ; α' , and priority $prty$.

²In ECharts [4], actions can neither produce triggering events nor change the values of shared variables. Thus, the order of concatenation of α and α' is unimportant here.

- **(Non-shared Transitions)** otherwise, if r is non-shared, add to R_+ a transition corresponding to r with event a , condition $c \wedge [\mathbf{ID} = \mathbf{M}_1]$, action α , and priority $prty$.

As an example, Figure 4 shows the resulting merge for the models of Figure 1 with respect to the relation ρ in Figure 3(c). The conditions shown in boldface in Figure 4 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 behaviour that is applicable only for clients subscribing to voicemail.

Two points need to be noted about our merge construction: (1) The construction requires that states be either atomic or superstates (OR states) – as noted in Section 3, parallel states (AND states) are replaced by their semantically equivalent non-parallel structures before merge. To keep the structure of the merged model as close as possible to the input models, non-shared parallel states can be exempted from this replacement when their descendants are all non-shared too. Such parallel states (and all descendants thereof) can be lifted verbatim to the merge. (2) 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 to ensure that merges are behaviourally sound and deterministic. However, such a conservative approach may result in redundant transitions. These redundancies arise due to logical or unstated relationships between the events and conditions used in the input models. For example, in Figure 4, 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) , it is equal 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) by `subscriber?Ack[ID=basic]`, and merge the two out-going transitions from (s_3, t_2) into one 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 Evaluation

The ultimate evaluation of our work is whether developers faced with model management tasks find our approach helpful. In some contexts, developers may find it relatively easy to identify matches by hand, for example if the models are small, and the developers are very familiar with them. Our approach to matching is valuable if it offers a quick way to identify appropriate matches with reasonable accuracy, 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. On the other hand, computing merge by hand is always likely to be laborious; our approach to merge is therefore useful if it produces semantically correct results and scales well.

Here, we present some initial steps to evaluate our work. First, we discuss the complexity of our Match and Merge operators, to show that they scale. We assess our Match operator by measuring the accuracy of the relations it produces, in comparison with the assessment of a domain expert. We assess our Merge operator by proving that it preserves the behavioural properties of the input models. In the longer term, we plan to conduct more extensive user evaluations to determine the value of our approach.

We have implemented our Match operator and have used it for the evaluation described in here. We have also developed a Merge tool, TReMer [28], for merging state-machines with respect to a given correspondence relation between them. TReMer implements the procedure in Section 5.2 and can merge Statecharts models stored as XML files, but currently lacks a user interface for displaying hierarchical models. We have successfully used TReMer for merging variant telecom features.

6.1 Complexity

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. 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 behavioural matching – $O(c \times m_1 \times m_2)$, where c is the maximum allowed number of iterations for the behavioural matching algorithm.

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 $O(m_1 \times m_2)$.

6.2 Accuracy 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 would 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 metrics, namely *precision*, and *recall*, to capture this intuition. Precision measures quality (i.e., 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., low number of false nega-

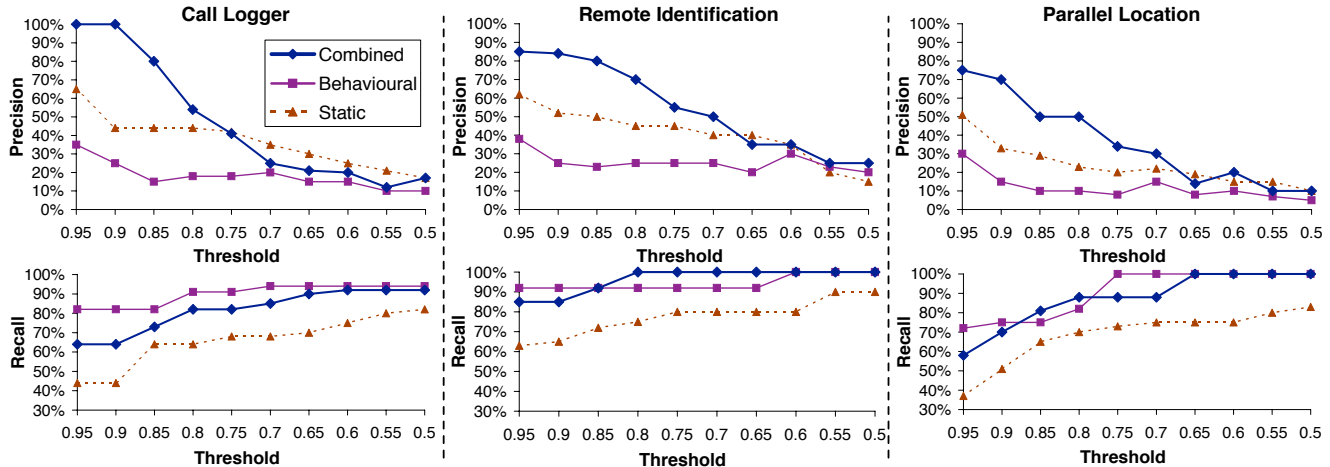


Figure 5. Results of static, behavioural, and combined matching.

tives) and is the ratio of the correct matches found to the total number of all correct matches. For example, if our matcher produces the relation in Figure 3(b) and the desired relation is Figure 3(c), the precision and recall is $8/14$ and $8/8$, 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. The Software Engineering literature suggests that for information retrieval tasks, users are willing to tolerate a small decrease in precision if it can bring about a comparable increase in recall [10]. We expect this to be true for model matching, especially for larger models: it is easier for users to remove incorrect matches rather than find missing ones. On the other hand, precision should not be too low. A precision less than 50% indicates that more than half of the found matches are wrong. In the worse case, it may take users more effort to remove incorrect matches and find missing correct matches than to do the matching manually.

We evaluated the precision and recall of our Match operator by applying it to a set of Statecharts 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 in Section 1.1. Simplified versions of the variants of this feature were shown in Figure 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

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

Table 1. Characteristics of the studied models.

second one has 19 states and 31 transitions. The correct relation (as identified manually by our domain expert) consists of 12 pairs of states.

To compare the overall effectiveness of static matching, behavioural matching, and their combination, we compute their precision and recall for thresholds³ ranging from 0.95 down to 0.5. The results are shown in Figure 5.

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. Behavioural 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 behavioural matching on their own. This indicates that static and behavioural matching are filtering out each other’s false positives. Recall remains high in the combined approach, as static matching and behavioural matching find many complementary high-quality matches.

Table 2 shows the precision-recall tradeoff points, which we believe to be reasonable for the studied examples. As shown in the table, for thresholds between 0.75 and 0.85, our combined matcher achieved a precision of more than 50% and a recall of more than 80%. We anticipate that for thresholds between 0.7 and 0.9 our technique would have acceptable precision and recall; however, a more decisive answer to this question requires further evaluation.

³Recall that threshold is the cutoff value used for determining the correspondence relation from the similarity degrees (see Section 4.4).

Feature	Threshold	Precision	Recall
Call Logger	0.80	54%	82%
Remote Identification	0.75	55%	100%
Parallel Location	0.85	51%	81%

Table 2. Tradeoff precisions and recalls.

6.3 Correctness of Merge

In [20], we have shown that our merge procedure is behaviour-preserving. The proof goes by first translating Statecharts models to their semantically equivalent Labelled Transition Systems (LTSs) [19], and then showing that the merge, when applied to LTSs, preserves branching temporal properties expressed in modal μ -calculus [12]. Intuitively, this is because: (1) The set of unguarded behaviours of the merge is a subset of the behaviours of the individual input models. Therefore, any universal μ -calculus property that holds over the input models also holds over the unguarded fragment of their merge. (2) Behaviours of the individual input models are present as either guarded or unguarded behaviours in their merge. Thus, the merge preserves all existential μ -calculus properties of the input models.

The merge includes, in either guarded or unguarded form, *every* behaviour of the input models. A change in the correspondence relation (ρ) does not cause any behaviours to be added to or removed from the merge, but may make some guarded behaviours unguarded, or vice versa. The use of parameterization for representing behavioural variabilities allows to generate behaviour-preserving merges for models that may even be inconsistent.

As noted in Section 3, our models have deterministic semantics, achieved by assigning priority labels to transitions. Our merge construction respects transition priorities and ensures that merges are deterministic as well.

Section 5 described our procedure for merging *pairs* of models. This can be extended to n -ary merges by iteratively merging a new input model with the result of a previous merge, with one minor modification: the reserved variable **ID** (in the merge procedure of Section 5.2) will 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.

7 Related Work

Matching. 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 [24]. State-of-the-art schema matchers, such as Protoplasm [3], 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 behavioural models.

Several approaches to matching have been proposed in software engineering. [14] employs heuristic reasoning for finding analogies between a problem description and already existing domain abstractions. [25] uses approximate graph matching for finding overlaps between concept graphs. [15] combines diagrammatic and syntactic heuristics for finding matches between architecture models. None of these were specifically designed for behavioural models and are either inapplicable or unsuitable for matching Statecharts models.

Our formulation of behavioural similarity (Section 4.2) is analogous to the Markovian notions of behavioural relations in [29]. Their goal is to define an overall distance measure between reactive processes, whereas our goal is to obtain a similarity measure between different Statecharts models for finding their correspondences.

Merging. Model merging spans several application areas. In database design, merge is an important step for producing a schema capturing the data requirements of all stakeholders [2]. Software engineering deals extensively with model merging – several papers study the subject in specific domains including early requirements [27], static UML diagrams [1, 17, 13, 33], scenarios [32], and state-machines [26, 31]. [26] proposed a structural technique for merging state-machines without regard to their behaviours. In contrast, [31] provided an approach for behavioural merging of consistent state-machines without variabilities. These earlier approaches neither handle hierarchical notations nor address the question of reconciling the structural and behavioural aspects of state-machine merging.

Our merge operator makes use of parameterization for representing variabilities between different models. This is a common technique in software maintenance and product line engineering [8]. A different approach to dealing with variabilities is to originally treat them as inconsistencies [7, 31, 27]. This is more suitable for early stages of development where variabilities may be due to conceptual disagreements between stakeholders.

8 Conclusions and Future Work

We presented an approach to matching and merging of Statecharts. Our Match operator includes heuristics that use both static and behavioural properties to match pairs of states in the input models. Preliminary evaluations show that this combination produces higher precision than relying on static or behavioural properties alone. Our Merge operator produces a combined model in which variant behaviours 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 developed a proof-of-concept implementation of these operators.

While our preliminary evaluations demonstrate the effectiveness of our approach, its practical utility can only be as-

essed by more extensive user trials. The value of our tools are 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 Match operator can be used in a number of different ways. In Section 6, 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, and we plan to run further experiments to investigate this idea. Alternatively, a developer might prefer to assess the output of the Match operator by computing the Merge, and inspecting the resulting model for validity. In 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 whether this approach is feasible.

Finally, note that our evaluations depend on a subjective judgment about the correct match to be found. In practice, different developers may not agree on the correct way to match their models [18]. We plan to conduct empirical evaluations that take this subjectivity into account.

Our work has a number of limitations that we plan to investigate further. As noted in Section 5.2, shared parallel states are replaced with their semantically equivalent non-parallel structures. This may result in discontinuities between the conceptual structuring of the merge and that of the input models when parallel states have many substates. In our telecom models, parallel states have no more than a few substates each (less than five); therefore, the merged models still retained the essential structure of the input models. An alternative approach to handling parallel states may be needed for domains that make more extensive use of parallelization. Moreover, the Statecharts models studied in Section 6 did not communicate through message passing or shared variables. It is interesting to see if the semantics of communications between models can help simplify our matching and merging procedures.

ECharts have a number of advanced features including transitions with multiple parallel source states and transitions with history targets. Since match is a heuristic process, we can either ignore these or find an approximating representation for them. We have not yet investigated how such features affect the accuracy of our Match operator. Our Merge operator can handle these features as long as they are non-shared. However, it currently does not provide explicit support for these features.

The work reported here is part of a larger and ongoing project on model management and its applications in software engineering. An early vision of this project was presented in [6]. Our main direction for future work 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.

Acknowledgments. We are grateful to Thomas Smith for help with our analysis of telecom features. We thank Mihaela Gheorghiu, Rick Salay, and the anonymous ICSE reviewers for their insightful comments. Financial support was provided by Bell Canada (through the Bell University Labs), OGS, NSERC, and an IBM Eclipse grant.

References

- [1] M. Alanen and I. Porres. Difference and union of models. In *UML*, pages 2–17, 2003.
- [2] P. Bernstein. Applying model management to classical meta data problems. In *CIDR*, pages 209–220, 2003.
- [3] P. Bernstein, S. Melnik, M. Petropoulos, and C. Quix. Industrial-strength schema matching. *SIGMOD Record*, 33(4):38–43, 2004.
- [4] G. Bond. An introduction to ECharts: The concise user manual. Technical report, AT&T, 2006. Available at: <http://echarts.org>.
- [5] G. Bond, E. Cheung, K. Purdy, P. Zave, and J. Ramming. An open architecture for next-generation telecommunication services. *ACM Trans. Inter. Tech.*, 4(1):83–123, 2004.
- [6] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *Wkshp. on Global Integrated Model Mgmt.*, 2006.
- [7] S. Easterbrook and M. Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *ICSE*, pages 749–750, 2001.
- [8] H. Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures*. Addison Wesley, first edition, 2004.
- [9] D. Harel and M. Politi. *Modeling Reactive Systems With Statecharts : The Statechart Approach*. McGraw Hill, 1998.
- [10] J. H. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements tracing via information retrieval. In *RE*, pages 138–147, 2003.
- [11] M. Jackson and P. Zave. Distributed feature composition: a virtual architecture for telecommunications services. *IEEE TSE*, 24(10):831–847, 1998.
- [12] D. Kozen. Results on the propositional μ -calculus. *TCS*, 27, 1983.
- [13] K. Letkeman. Comparing and merging uml models in ibm rational software architect. Technical report, IBM, 2006.
- [14] N. Maiden and A. Sutcliffe. Exploiting reusable specifications through analogy. *CACM*, 35(4):55–64, 1992.
- [15] D. Mandelin, D. Kimelman, and D. Yellin. A Bayesian approach to diagram matching with application to architectural models. In *ICSE*, pages 222–231, 2006.
- [16] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.
- [17] A. Mehra, J. Grundy, and J. Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *ASE*, pages 204–213, 2005.
- [18] S. Melnik. *Generic Model Management: Concepts And Algorithms*, volume 2967 of *LNCIS*. Springer, 2004.
- [19] R. Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1989.
- [20] S. Nejati. Statecharts merging: Mathematical underpinnings. Technical Report CSRG-546, U. of Toronto, 2007.
- [21] R. D. Nicola, U. Montanari, and F. Vaandrager. Back and forth bisimulations. In *CONCUR*, pages 152–165, 1990.
- [22] J. Niu, J. M. Atlee, and N. A. Day. Template semantics for model-based notations. *IEEE TSE*, 29(10):866–882, 2003.
- [23] T. Pedersen, S. Patwardhan, and J. Michelizzi. WordNet: Similarity - measuring the relatedness of concepts. In *AAAI*, pages 1024–1025, 2004.
- [24] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.
- [25] K. Ryan and B. Mathews. Matching conceptual graphs as an aid to requirements re-use. In *RE*, pages 112–120, 1993.
- [26] M. Sabetzadeh and S. Easterbrook. Analysis of inconsistency in graph-based viewpoints: A category-theoretic approach. In *ASE*, pages 12–21, 2003.
- [27] M. Sabetzadeh and S. Easterbrook. View merging in the presence of incompleteness and inconsistency. *RE J.*, 2006.
- [28] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik. TRMer: A tool for relationship-driven model merging. In *FM*, 2006. Demo.
- [29] O. Sokolsky, S. Kannan, and I. Lee. Simulation-based graph similarity. In *TACAS*, pages 426–440, 2006.
- [30] G. Spanoudakis and A. Finkelstein. Reconciling requirements: A method for managing interference, inconsistency and conflict. *Annals of Soft. Eng.*, 3:433–457, 1997.
- [31] S. Uchitel and M. Chechik. Merging partial behavioural models. In *FSE*, pages 43–52, 2004.
- [32] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *ICSE*, pages 314–323, 2000.
- [33] A. Zito, Z. Diskin, and J. Dingel. Package merge in UML 2: Practice vs. theory? In *MoDELS*, pages 185–199, 2006.