Matching and Merging of Variant Feature Specifications

Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave

***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 article, 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 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. 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, Behaviour preservation, Variability modelling, Parameterization.**

I. Introduction

Model-based development involves construction, integration, and maintenance of complex models. For large-scale projects, modelling is often a distributed endeavor involving multiple teams at different organizations and geographical locations. These teams build multiple inter-related 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].

Shiva Nejati and Mehrdad Sabetzadeh are with Simula Research Laboratory, Lysaker, Norway. Email: {shiva,mehrdad}@simula.no.

Marsha Chechik and Steve Easterbrook are with the Department of Computer Science, University of Toronto, Toronto, ON, Canada. Email: {chechik,sme}@cs.toronto.edu.

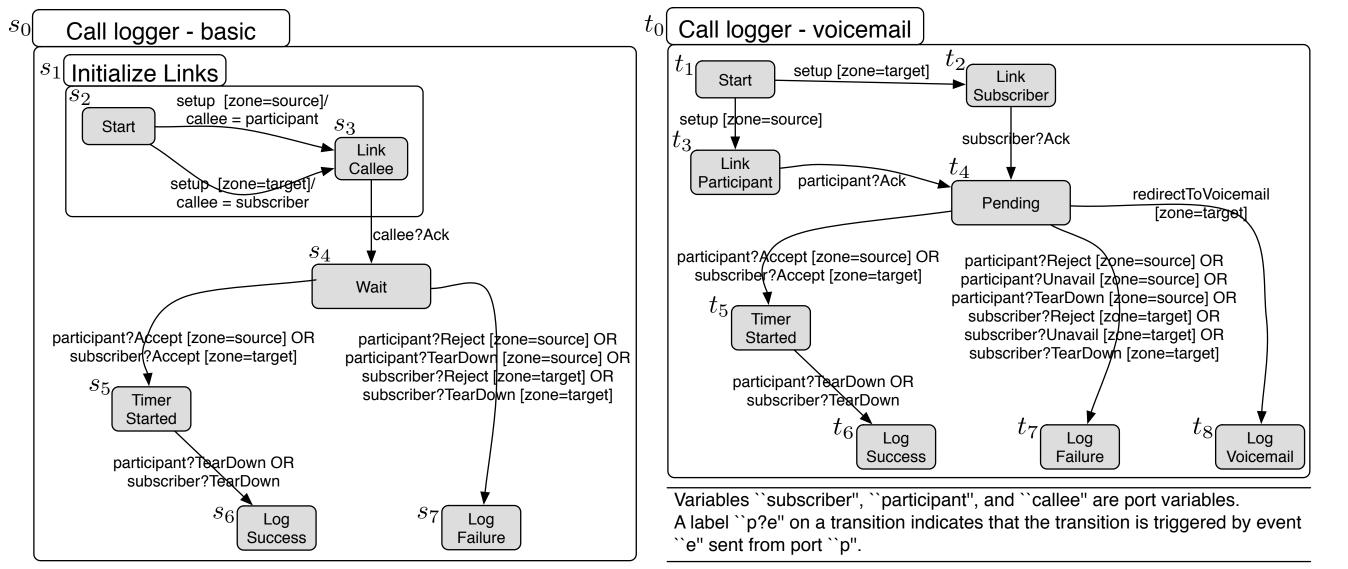
Pamela Zave is with AT&T Laboratories–Research, Florham Park, NJ, USA. Email: pamela@research.att.com

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

In contrast, research on behavioural 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 behavioural 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 modellers 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 article, we present an approach to matching and



s_0 Call logger - basic
s_1 Initialize Links
s_2
Start
setup [zone=source]/ callee = participant
s_3 Link Callee
setup [zone=target]/ callee = subscriber
callee?Ack
s_4 Wait
participant?Accept [zone=source] OR subscriber?Accept [zone=target]
participant?Reject [zone=source] OR participant?TearDown [zone=source] OR subscriber?Reject [zone=target] OR subscriber?TearDown [zone=target]
s_5 Timer Started
participant?TearDown OR subscriber?TearDown
s_6 Log Success
s_7 Log Failure
t_0 Call logger - voicemail
t_1 Start
setup [zone=target]
t_2 Link Subscriber
setup [zone=source]
subscriber?Ack
t_3 Link Participant
participant?Ack
t_4 Pending
redirectToVoicemail [zone=target]
participant?Accept [zone=source] OR subscriber?Accept [zone=target]
participant?Reject [zone=source] OR participant?Unavail [zone=source] OR participant?TearDown [zone=source] OR subscriber?Reject [zone=target] OR subscriber?Unavail [zone=target] OR subscriber?TearDown [zone=target]
t_5 Timer Started
participant?TearDown OR subscriber?TearDown
t_6 Log Success
t_7 Log Failure
t_8 Log Voicemail

Variables ``subscriber", ``participant", and ``callee" are port variables.
A label ``p?e" on a transition indicates that the transition is triggered by event ``e" sent from port ``p".

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.

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

merging variant feature specifications described as Statechart models. Merging combines structural and semantic information present in the Statechart models and ensures that their behavioural 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 behavioural 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.

A. 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,

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 behavioural properties of the models in Figure 1: **P1** represents an overlapping behaviour, and **P2** – a non-overlapping one.

the address location where a call is answered, 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. 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 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 feature. For example, consider the two properties described in Figure 2. Property **P1** holds in both variants shown in Figure 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 behaviours 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 behavioural variation between the variants shown in Figure 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.

B. Contributions of this article

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 non-shared parts of the input models in their merge? What properties of the input models should be preserved by their merge? In this article, we address these questions for models expressed as Statecharts. This article 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 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 models.
- A description of a Merge operator for Statechart models. We provide a procedure for constructing behaviour-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 article is organized as follows. Section II provides an overview of our Match and Merge operators. Section III outlines our assumptions and fixes notation. Section IV introduces our Match operator, and Section V – our Merge operator. Section VI describes tool support for the two operators. Section VII presents an evaluation of effectiveness for the Match operator, and Section VIII assesses the soundness of the Merge operator. Section IX compares our contributions with related work and discusses the results presented in this article. Finally, Section X concludes the article.

II. Overview of Our Approach

Figure 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.

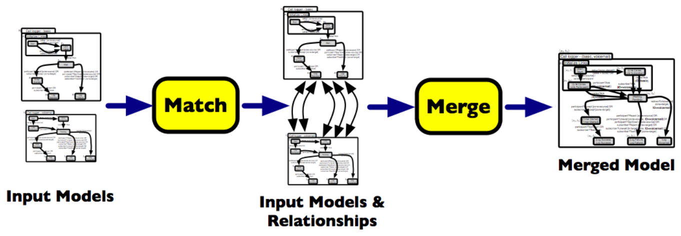


Fig. 3. A framework for integrating variant feature specifications.

Our ultimate goal is to provide automated tool support for the framework in Figure 3. Among these two operators, Match has a heuristic nature. Since models are developed independently, we can never be entirely sure about 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 behavioural. Static heuristics use structural and textual 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.

To obtain a satisfactory correspondence relation, we use a combination of static and behavioural heuristics. Our Match operator produces a correspondence relation between states in the two models. For the models in Figure 1, it may yield the correspondence relation shown in Figure 8(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 8(d).

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:

1) preserves the behavioural properties of the input models. Figure 10 shows the merge of the models of Figure 1 with respect to the relation in Figure 8(d). This merge is behaviour-preserving. That is, any behaviour of the input models is preserved in the merge model (either

through shared or non-shared behaviours). For example, the property **P1** in Figure 2 that shows an overlapping behaviour between the models in Figure 1 is preserved in the merge as a shared behaviour (denoted by the path from state (s_4, t_4) to (s_6, t_6)).

2) distinguishes between shared and non-shared behaviours of the input models by attaching appropriate guard conditions to non-shared transitions. In the merge, non-shared transitions are guarded by boldface conditions representing the models they originate from. For example, the property **P2** in Figure 2 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.
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.

III. Assumptions and Background

The Statechart language [22] is a common notation for describing hierarchical state machines and is a de-facto standard for specifying intra-object behaviours 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 $s \xrightarrow{e[c]/\alpha}_{prty} s'$. 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 Figure 4(a), 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 Figure 4(a) is shown in Figure 4(b). In the Statechart of the basic call logger model in Figure 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 $\{\text{setup}, \text{Ack}, \text{Accept}, \text{Reject}, \text{TearDown}\}$, and the set V of variables is $\{\text{callee}, \text{zone}, \text{participant}, \text{subscriber}\}$. The only actions in Figure 1 are callee=participant and callee=subscriber. These actions assign values participant and subscriber to the variable callee, respectively.

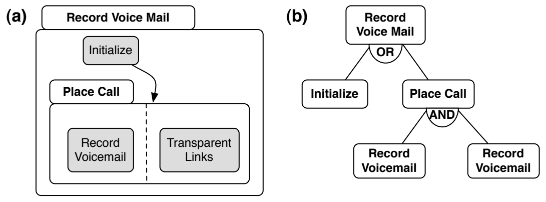


Fig. 4. Parallel Statecharts: (a) An example, and (b) the hierarchy tree corresponding to (a).

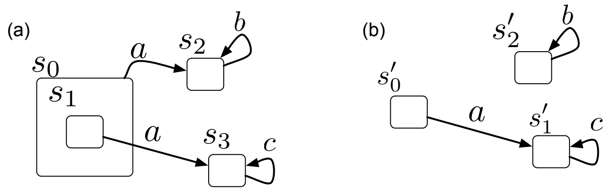


Fig. 5. (a) Prioritizing transitions to eliminate non-determinism in ECharts: Transition $s_1 \to s_3$ has higher priority than transition $s_0 \to s_2$, and (b) the flattened form of the Statecharts in (a).

Implementations of the Statechart language differ on how they define the semantics of inter- and intra-machine communication, and how they resolve non-determinism 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 intra-machine 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 micro-steps coincide in ECharts.
- **Non-determinism.** 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 Figure 5(a), states s_0 and s_1 have transitions labelled a to two different states, s_2 and s_3, respectively. This makes the semantics of this Statechart model non-deterministic because it is not clear which of the transitions, $s_0 \to s_2$ or $s_1 \to s_3$, should be taken upon receipt of the event a. In ECharts, certain types of non-determinism are resolved by assigning global priorities (using $prty$) to transitions that have the same event and condition. For example, in Figure 5(a), 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 Figure 1 are already deterministic, i.e., any external event triggers at most one transition in them. Thus, no further prioritization is required[1].

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

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

semantic variation points are resolved in a particular Statechart implementation. Our proof for semantic preservation of Merge (see Appendix XI-C) can carry over to other dialects.

In addition, we make the following assumptions on how behavioural models are developed in our context. Let $M_1 = (S_1, \hat{s}, <^1_h, E_1, V_1, R_1)$ and $M_2 = (S_2, \hat{t}, <^2_h, E_2, V_2, R_2)$ 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 Figure 1 are callee=participant and callee=subscriber. These actions do not cause any interaction between the Statechart models in Figure 1. Hence, the models in Figure 1 are non-interacting. For a discussion on distinctions between models with interacting vs. overlapping behaviours, see Section IX.

IV. Matching Statecharts

Our Match operator (Figure 6) uses a hybrid approach combining static matching, $\mathcal{S}$ (Section IV-A), and behavioural matching, $\mathcal{B}$ (Section IV-B). 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 $\mathcal{D}$. Behavioural matching ($\mathcal{B}$) generates similarity degrees between states based on their behavioural 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 behavioural heuristics to generate the overall similarity degrees between states (Section IV-C). Given a similarity threshold, we can then determine a correspondence relation ρ over the states of the input models (Section IV-C).

A. Static Matching

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

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

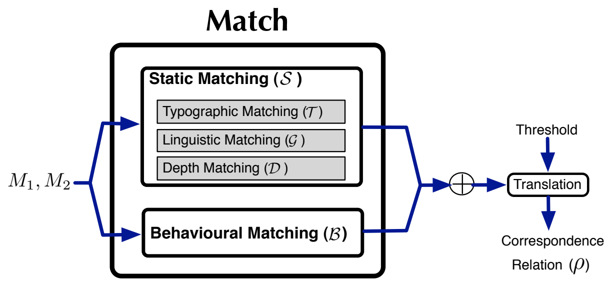

Match
Static Matching ($\mathcal{S}$)
Typographic Matching ($\mathcal{T}$)
Linguistic Matching ($\mathcal{G}$)
Depth Matching ($\mathcal{D}$)
Behavioural Matching ($\mathcal{B}$)
M_1, M_2
Threshold
Translation
Correspondence Relation (ρ)

Fig. 6. Overview of the Match operator.

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., N = 3). For example, the result of trigram matching for some of the name labels of the states in Figure 1 is as follows:

trigram("Wait", "Pending")	=	0.0
trigram("Log Success", "Log Failure")	=	0.21
trigram("Log Success", "Log Success")	=	1.0
trigram("Link Callee", "Link Participant")	=	0.18

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 Figure 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 nodes on each side

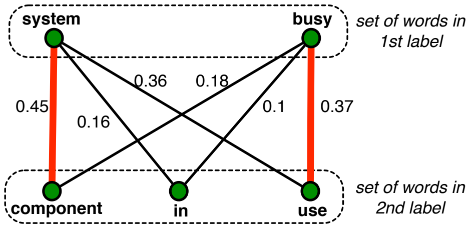


Fig. 7. Weighted bipartite graph induced by a pair of name labels ("system busy" and "component in use")

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". Figure 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 Figure 1, $depth(s_2)$ is 2 and $depth(t_1)$ is 1, and $\mathcal{D}(s_2, t_1) = 0.5$.

B. Behavioural Matching

Behavioural matching ($\mathcal{B}$) provides a measure of similarity between the behaviours of different states. Our behavioural matching technique draws on the notion of bisimilarity between state machines [31]. Bisimilarity provides a natural way to characterize behavioural 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-labelled transitions; and are (forward) dissimilar otherwise.

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

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 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' labelled a, or s' is 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 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. 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 $i-1$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 $\mathcal{B}^{i-1}(s,t)$:

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

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

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 behavioural similarity, $\mathcal{B}$, as the maximum of forward behavioural and backward behavioural matching.

C. 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 is available [4]. Since such data was 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 IV-A). Figure 8(a) illustrates $\mathcal{C}$ for the models in Figure 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%, we obtain the correspondence relation ρ shown in Figure 8(b).

	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.

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 Figure 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) behavioural 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 Figure 8(b). 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 V-A, the relation ρ may need to be further revised before merging to ensure that the resulting merged model is well-formed.

V. 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 V-A). These checks ensure that merging M_1 and M_2 using ρ results in a well-formed (i.w., structurally sound) Statechart model. If the checks fail, a subset of ρ violating the checks is identified.

A. 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:

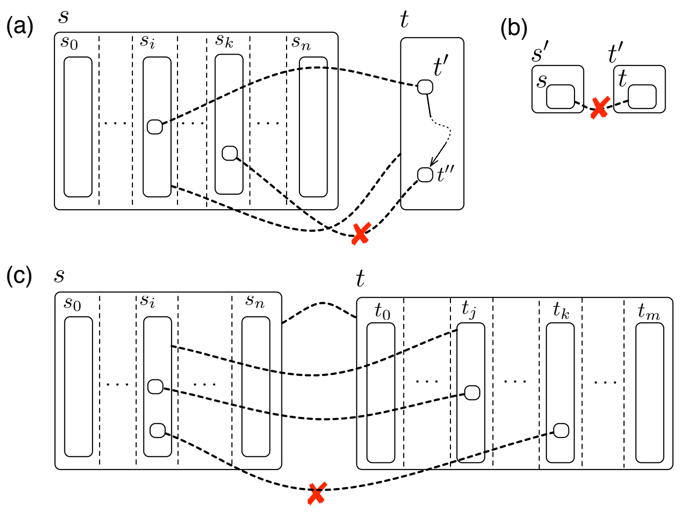


Fig. 9. Example violation of sanity checks: (a) and (c) violations of AND-states integrity rules, and (b) violation of relational adequacy.

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 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 $\hat{s}$ (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 sub-states of s. Then, if ρ maps a proper descendant of $s_i; (0 \leq i \leq n)$ to a state t', (1) there must be some ancestor of t' mapped to s_i by ρ, and (2) 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 Figure 9(a).
      iii) Let s and t be AND-states, let $s_0, \ldots, s_n$ be parallel sub-states of s, and let $t_0, \ldots, t_m$ be parallel sub-states 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 Figure 9(c).

   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 non-parallel 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 Figure 8(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. 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 Figure 8(d), passes all sanity checks and can be used for merge. Consider the example shown in Figure 9(b). 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.

B. Merge Construction

Let M_1 and M_2 be Statechart models. To merge them, 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, 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 behaviour [33]. For example, $z = x$ and $y = x$ are two independent actions, but $x = y + 1$ and $z = x$ are not independent. r is *non-shared* otherwise.

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 [34]: 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.

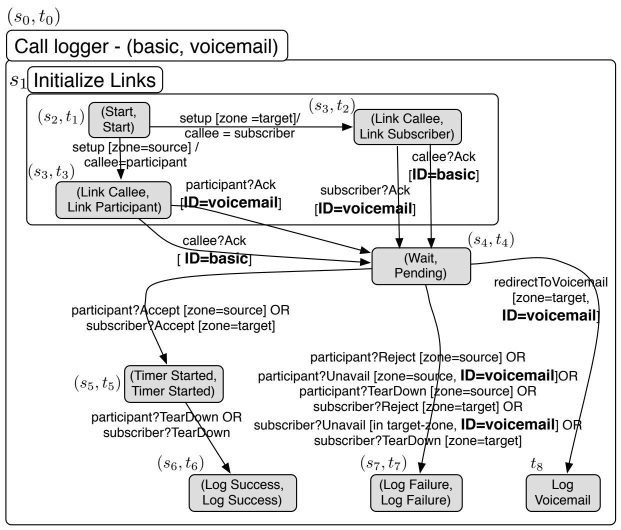

(s_0, t_0)
Call logger - (basic, voicemail)
s_1 Initialize Links
(s_2, t_1) (Start, Start)
setup [zone =target]/ callee = subscriber
(s_3, t_2) (Link Callee, Link Subscriber)
setup [zone=source] / callee=participant
(s_3, t_3) (Link Callee, Link Participant)
participant?Ack [ID=voicemail]
subscriber?Ack [ID=voicemail]
callee?Ack [ID=basic]
callee?Ack [ID=basic]
(s_4, t_4) (Wait, Pending)
redirectToVoicemail [zone=target, ID=voicemail]
participant?Accept [zone=source] OR subscriber?Accept [zone=target]
(s_5, t_5) (Timer Started, Timer Started)
participant?Reject [zone=source] OR
participant?Unavail [zone=source, ID=voicemail]OR
participant?TearDown [zone=source] OR
subscriber?Reject [zone=target] OR
subscriber?Unavail [in target-zone, ID=voicemail] OR
subscriber?TearDown [zone=target]
participant?TearDown OR subscriber?TearDown
(s_6, t_6) (Log Success, Log Success)
(s_7, t_7) (Log Failure, Log Failure)
t_8 Log Voicemail

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

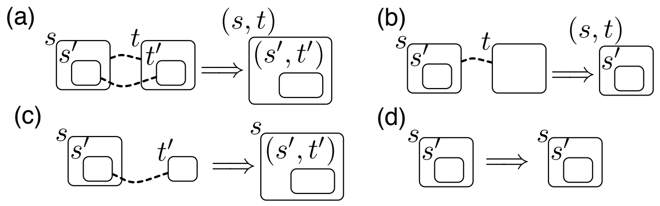


Fig. 11. Merging different OR-state patterns (note that part (d) refers to the case where both s and s' are non-shared).

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.

- **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.** ($<_h^+$) The hierarchy tree $<_h^+$ of $M_1 +_\rho M_2$ is computed as follows:
  (I) **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.
  - 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 $M_1 +_\rho M_2$ (see Figure 11(a)).
    * otherwise, if s' is non-shared, make s' a child of (s, t) in $M_1 +_\rho M_2$ (see Figure 11(b)).
  - otherwise, if s is non-shared
    * if s' is mapped to a state t' by ρ, make (s', t') a child of s in $M_1 +_\rho M_2$ (see Figure 11(c)).
    * otherwise, if s' is non-shared, make s' a child of s in $M_1 +_\rho M_2$ (see Figure 11(d)).

  (II) **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 sub-states of s.
  - Let t be an AND-state in M_2, and let $t_0, \ldots t_m$ be parallel sub-states of t. If s is mapped to t by ρ, make (s, t) an AND-state in $M_1 +_\rho M_2$ and
    * 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 Figure 12(a)).
    * otherwise, if s_i $(0 \leq i \leq n)$ is non-shared, make s_i a child of (s, t) in $M_1 +_\rho M_2$ (see Figure 12(b)).
  - otherwise, if s is non-shared
    * 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 Figure 12(c)).
    * otherwise, if s_i $(0 \leq i \leq n)$ is non-shared, make s_i a child of s in $M_1 +_\rho M_2$ (see Figure 12(d)).
- **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, action α (if $\alpha = \alpha'$) or action $\alpha; \alpha'$ (if $\alpha \neq \alpha'$), and priority $prty$.
    Note that according to the definition of shared tran-

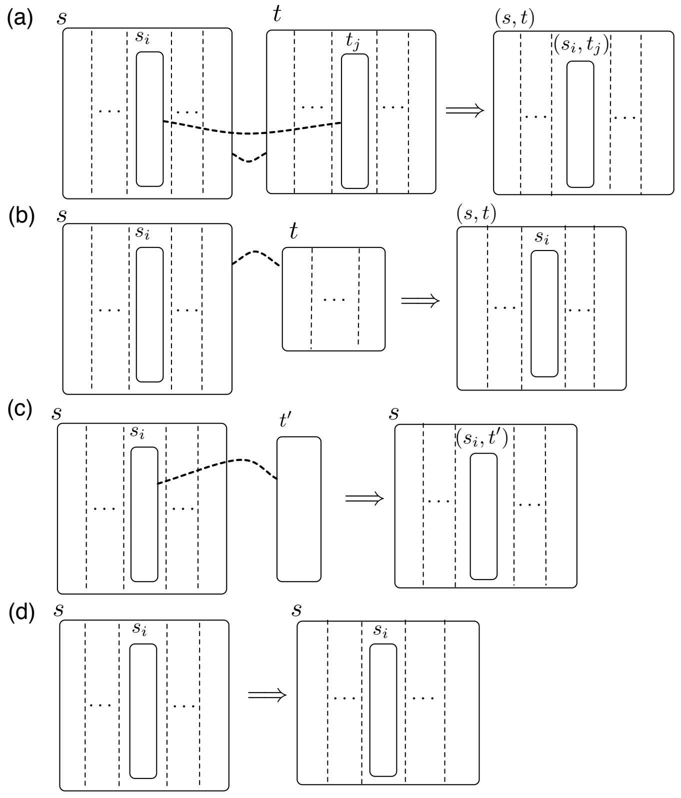


Fig. 12. Merging different AND-state patterns (note that part (d) refers to the case where both s and s' are non-shared).

sitions, α and α' are independent. Moreover, based on our assumptions in Section III, 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.

– (**Non-shared Transitions**) otherwise, if r is non-shared, add to R_+ a transition corresponding to r with event a, condition $c \wedge [\mathbf{ID} = M_1]$, action α, and priority $prty$.

As an example, Figure 10 shows the resulting merge for the models of Figure 1 with respect to the relation ρ in Figure 8(d). The conditions shown in boldface in Figure 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 behaviour 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 behaviourally sound and do not introduce additional non-determinism. 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 Figure 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) 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.

VI. Tool Support

We have implemented our Match and Merge operators, respectively described in Sections IV and V, 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 article 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* [1]. Such a 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.

Figure 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.

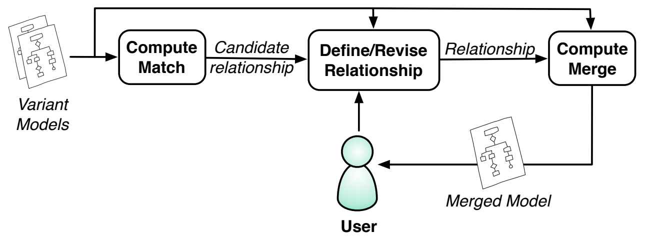


Fig. 13. Tool support overview.

In the remainder of this section, we illustrate our tool using the variant models in Figure 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 Figure 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 Figure 8(d). Note that the tool represents hierarchical states using an arrow with a hollow tail from each sub-state to its immediate super-state. For example, the arrow from start to initialize_Link (right side of Figure 14) indicates that initialize_Link is the immediate super-state of start. The merge computed by the tool with respect to the relationship defined above is shown in Figure 15. As seen from the figure, non-shared behaviours are guarded by conditions denoting the input model exhibiting those behaviours.

VII. 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 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.

A. 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 IV-B) 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.

B. 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 Figure 8(b) and the desired relation is as shown in Figure 8(d), the precision and recall are $8/14$ (57%) and $8/8$ (100%), 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 Figure 8(d): when our matcher produces the relation in Figure 8(b), 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 Figure 1. Of course, precision should not be too low – anything under 50% 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!

TABLE I
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

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 article 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 I-A. 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 I, 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, behavioural 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 Figure 16. As stated earlier in Section IV-C, threshold refers to the cutoff value used for determining the correspondence relation from the similarity degrees. The three diagrams at the top of Figure 16 represent the precision values for the three case studies in Table I, and the three diagrams at the bottom of that figure 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 behavioural matchings for the remote identification

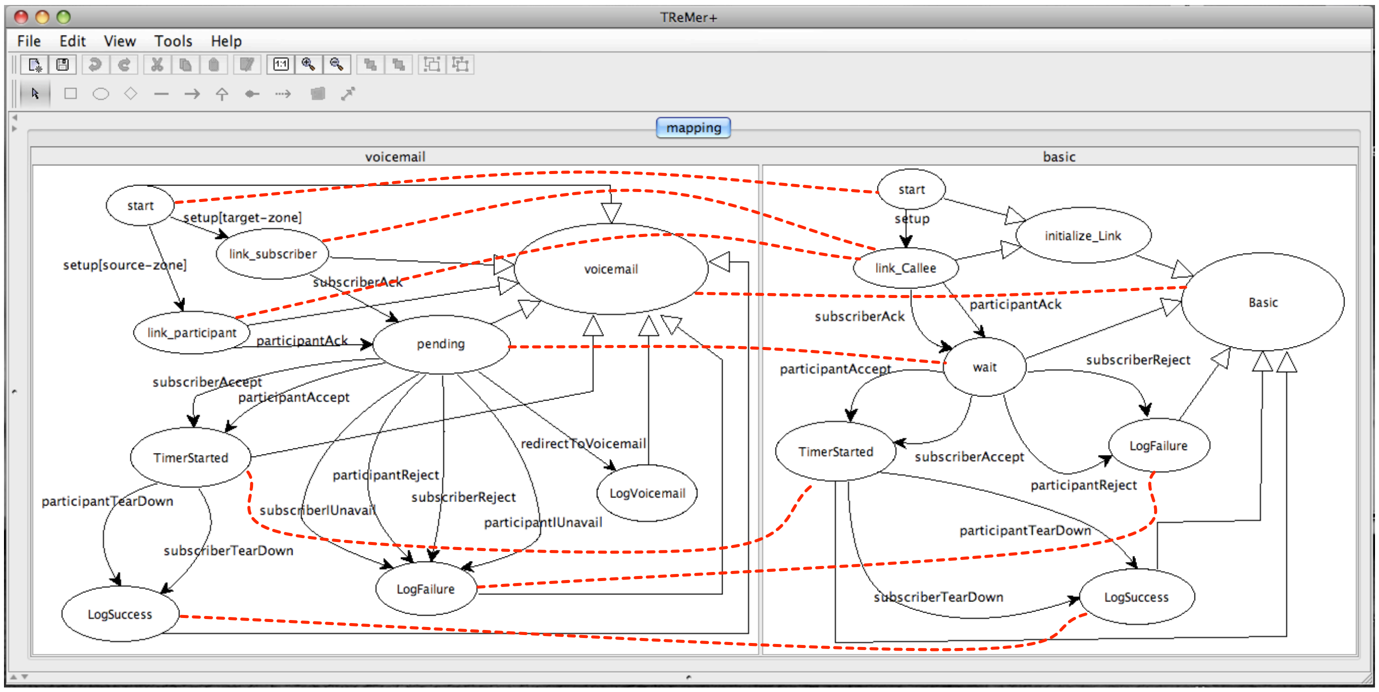


Fig. 14. Relationship between the models in Figure 1.

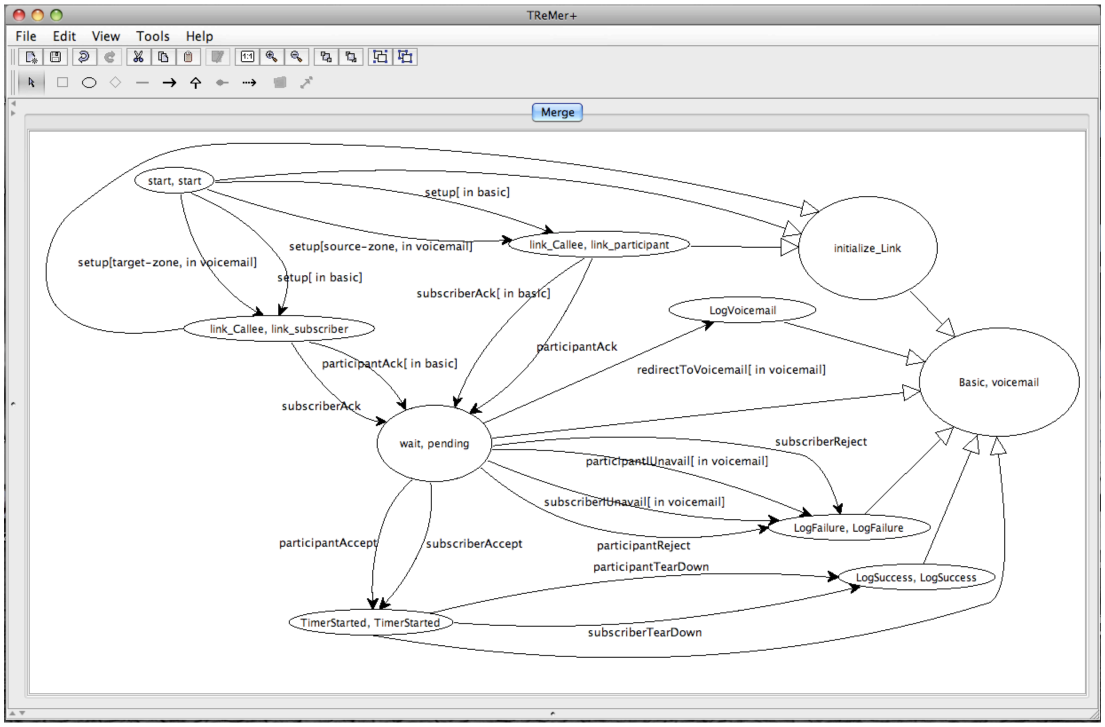


Fig. 15. Merge with respect to the relationship in Figure 14.

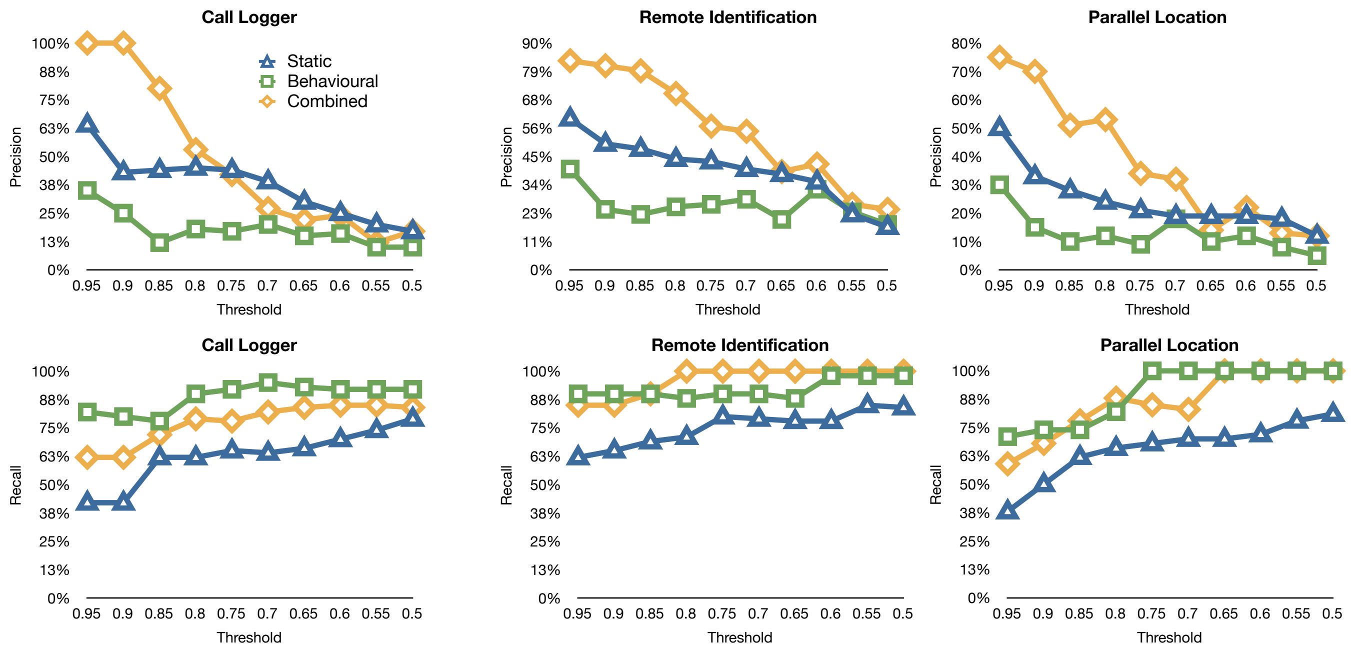


Fig. 16. Results of static, behavioural, and combined matching.

case study are 81%, 50%, and 24%, 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. 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 complimentary high-quality matches.

The results in Figure 16 show that for each of the studied models, our combined matcher can achieve high precision (above 75%) for some thresholds, and high recall (above 95%) 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.

$$\text{F}_2\text{-Measure} = \frac{3 \times \text{Precision} \times \text{Recall}}{(2 \times \text{Precision}) + \text{Recall}}$$

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

Table II (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%, and their corresponding recall values – between 62% and 100%.

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 behavioural 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 behavioural 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 V-A. For example, applying these to the relation in Figure 8(b) results in the removal of the tuples (s_1, t_0) and (s_1, t_1) (see Figure 8(c)). This leads to an increase in precision from $8/14$ (57%) to $8/12$ (67%) and recall remains the same, i.e., $8/8$ (100%). Figure 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/behavioural/combined algorithms as in Figure 16. Compared to the results in Figure 16, this methodology yields a 5%-25% improvement in precision across the different thresholds. The last two columns in Table II represent

TABLE II
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

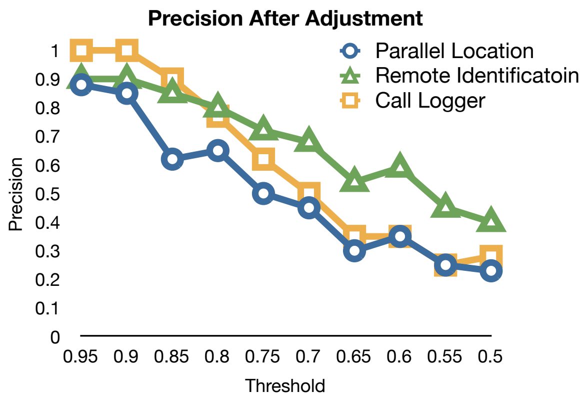


Fig. 17. Precisions rates for the case study models in Table I after applying Sanity checks in Section V-A.

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 do 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.

VIII. 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 behavioural properties of the input models.

A. 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 I). The time complexity of Merge is $O(m_1 \times m_2)$.

B. Correctness of Merge

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

Non-classical state machine formalisms have been previously defined to capture *partiality* [40]. Such models have two types of transitions: for describing definite and partial behaviours. In our work, in contrast, we use non-classical state machines to explicitly capture behavioural variabilities. Variant models differ on some of their behaviours, i.e., those that are non-shared, giving rise to variabilities. We use parameterized Statecharts to explicitly differentiate between shared behaviours, i.e., those that are common between all variants, and non-shared behaviours, i.e., those that differ from one variant to another. Specifically, in these Statechart models, transitions labelled by a condition on the reserved variable **ID** represent the non-shared behaviours, 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 behaviours, i.e., containing the transitions not labelled with **ID**, and $M^{nonshared} = (S, \hat{s}, <_h, E, V, R^{nonshared})$ is a Statechart representing non-shared behaviours, i.e., containing only the transitions with **ID**. We denote the set of both shared and non-shared 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 behaviours remain intact, but the partial behaviours may turn into definite or prohibited behaviours [40]. We define a new notion of refinement over parameterized Statechart models where (1) the non-shared behaviours are preserved through refinement, but the shared ones may turn into non-shared, and (2) the union of shared and non-shared behaviours does not change by refinement. That is, a model is more refined if it can capture more behavioural variabilities without changing the overall union of commonalities and variabilities. For example, the parameterized Statechart in Figure 10 refines both models in Figure 1 because it preserves all the behaviours of the models in Figure 1, and further, it captures more behavioural variabilities. The models in Figure 1 are parameterized Statecharts without non-shared behaviour.

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

1) The set of behaviours of M_2 is a subset of the set of behaviours of M_1. That is, as we refine, we do not lose any behaviour.
2) The set of shared behaviours of M_1 is a subset of the set of shared behaviours of M_2. That is, as we refine, we may increase behavioural 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 V-B. Then, $M_1 +_\rho M_2$ refines both M_1 and M_2.

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

(i) Behaviours of the individual input models, M_1 and M_2, are present as either shared, i.e., unguarded, or non-shared, i.e., guarded, behaviours in their merge, $M_1 +_\rho M_2$. Thus, the merge preserves all positive traces of the input models. For example, the positive behaviours $\mathbf{P_1}$ and $\mathbf{P_2}$ in Figure 2 are both preserved in the merge in Figure 10: $\mathbf{P_1}$ as an unguarded behaviours, and $\mathbf{P_2}$ an a guarded behaviour.

(ii) The set of shared, i.e., unguarded, behaviours of $M_1 +_\rho M_2$ is a subset of the behaviours of the individual input models, M_1 and M_2. Therefore, any behaviour absent from either input is absent from the unguarded fragment of their merge. In other words, any negative behaviour, i.e., safety property, that holds over the input models also holds over the unguarded fragment of their merge.

(iii) The guarded (non-shared) behaviours of the input models M_1 and M_2 are preserved in $M_1 +_\rho M_2$, i.e., merge preserves behavioural disagreements. But the unguarded (shared) behaviours of M_1 and M_2 may become non-shared in $M_1 +_\rho M_2$, i.e., merge can turn behavioural agreements into disagreements. For example, the transition t_4 to t_7 in Figure 1 represents an unguarded (shared) behaviour of the voicemail variant. But it turns into a guarded (non-shared) behaviour in the merge, as exemplified by the transition from (s_4, t_4) to t_8 in Figure 10.

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

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. For example, if we remove the tuple (s_7, t_7) from the correspondence relation ρ in Figure 8(d), the resulting merge is the model in Figure 18. The model in this figure still preserves every behaviour of the input models, but has more parameterized behaviours, 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 non-determinism is introduced. Section V 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 $\mathbf{ID}$ (in the merge procedure of Section V-B) 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.

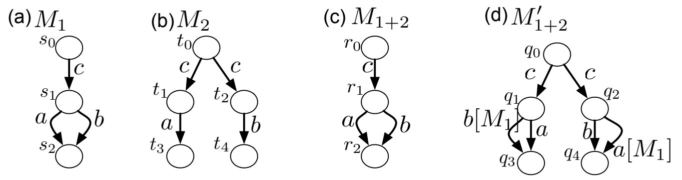


Fig. 19. (a) model M_1; (b) model M_2; (c) M_{1+2}: a possible merge of M_1 and M_2 that preserves their behaviours; and (d) M'_{1+2}: a possible merge of M_1 and M_2 that preserves their structure.

IX. Discussion

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

A. Structural vs. Behavioural 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 behaviour 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 behavioural models because model relationships are restricted to graph homomorphisms which are tools for preserving model *structure*, rather than *behavioural* properties.

We show the difference between structure-preserving and behaviour-preserving merges using a simple example. Consider the models M_1 and M_2 in Figures 19(a) and (b), 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 Figure 19(c) 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 behaviours of M_1 and M_2 because it collapses two behaviourally distinct states t_1 and t_2 into a single state r_1 in the merge. The model in Figure 19(d) 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 article: It can be shown that M'_{1+2}

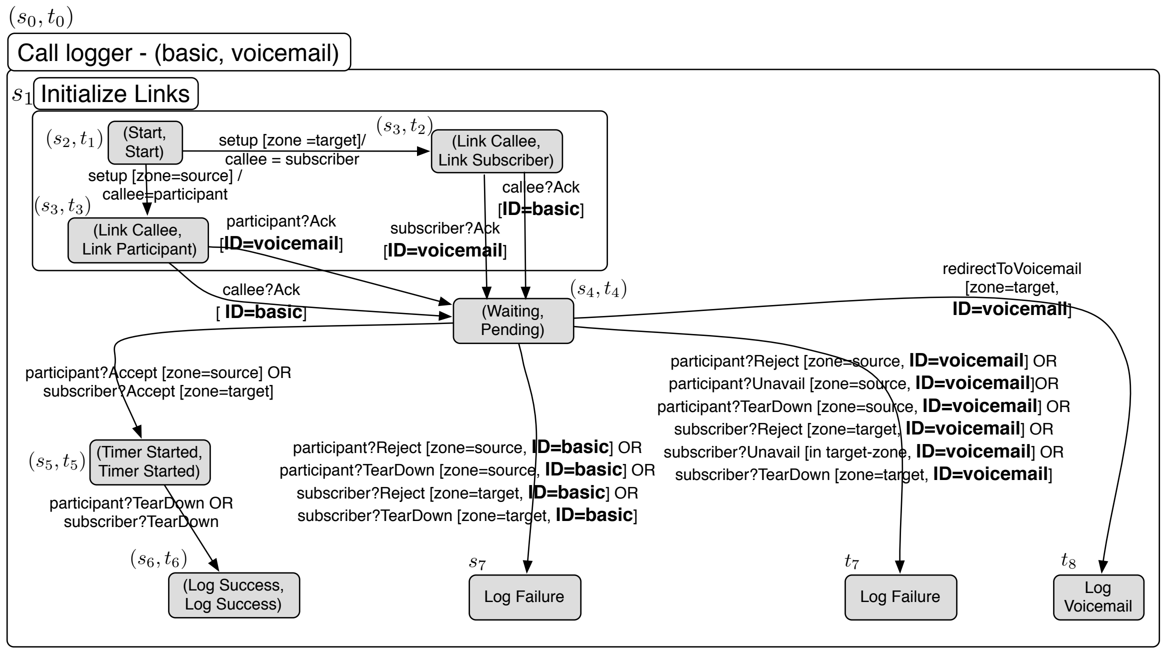

(s_0, t_0)
Call logger - (basic, voicemail)
s_1 Initialize Links
(s_2, t_1) (Start, Start)
setup [zone =target]/ callee = subscriber
(s_3, t_2) (Link Callee, Link Subscriber)
setup [zone=source] / callee=participant
(s_3, t_3) (Link Callee, Link Participant)
participant?Ack [ID=voicemail]
subscriber?Ack [ID=voicemail]
callee?Ack [ID=basic]
callee?Ack [ID=basic]
(s_4, t_4) (Waiting, Pending)
redirectToVoicemail [zone=target, ID=voicemail]
participant?Accept [zone=source] OR subscriber?Accept [zone=target]
participant?Reject [zone=source, ID=voicemail] OR participant?Unavail [zone=source, ID=voicemail]OR participant?TearDown [zone=source, ID=voicemail] OR subscriber?Reject [zone=target, ID=voicemail] OR subscriber?Unavail [in target-zone, ID=voicemail] OR subscriber?TearDown [zone=target, ID=voicemail]
participant?Reject [zone=source, ID=basic] OR participant?TearDown [zone=source, ID=basic] OR subscriber?Reject [zone=target, ID=basic] OR subscriber?TearDown [zone=target, ID=basic]
(s_5, t_5) (Timer Started, Timer Started)
participant?TearDown OR subscriber?TearDown
(s_6, t_6) (Log Success, Log Success)
s_7 Log Failure
t_7 Log Failure
t_8 Log Voicemail

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

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 behaviour preservation but is not supported by the merge approaches that are based on graph homomorphisms.

B. Merging Models with Behavioural Discrepancies

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

C. 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 article 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.

D. Overlapping, Interacting and Cross-Cutting Behavioural 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 article focuses on merging a collection of inter-related models when relationships describe *overlaps* between the models' behaviours. 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 behaviour (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 article was on relationships that capture overlaps between model behaviours, and not on situations where model relationships describe interactions or cross-cutting aspects.

E. Model Matching Techniques

Approaches to model matching can be *exact* or *approximate*. Exact matching is concerned with finding structural or behavioural conformance relations between models. Graph homomorphisms are examples of the former, whereas simulation and bisimulation relationships – of 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 behaviours 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 IV-A) 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. 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 behavioural models and are either inapplicable or unsuitable for matching Statechart models.

Some matching approaches deal with behavioural 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 modelling 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 both to 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 behavioural 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 behavioural conformance have been proposed to capture the behavioural 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 behaviours 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 behavioural similarity (e.g., [61], [62]). Our formulation of behavioural similarity (Section IV-B) 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 behavioural similarity.

F. 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 behavioural models and are either inapplicable or unsuitable for matching Statechart models. We compared our work with existing approaches for merging behavioural models in Section IX-B.

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 meta-models. 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 modelling tasks such as merging using model transformation languages defined between different meta-models. Despite their versatility, the current version 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 behavioural models.

In this article, we focused on the application of behavioural merge as a way to reconcile models developed independently.

Behavioural 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 modelling have been proposed in software maintenance and product line engineering. For example, [70] provides an elaborate view of modelling 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 modelling behavioural 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 behaviour-preserving refactoring of product line models from a set of existing (legacy) model variants [75].

G. Handling Additional Statechart Features

Our approach to Match and Merge can be systematically extended to handle Statchart 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 V-A) so that the new features in the original models are properly lifted to the merge, and (3) to modify the Merge algorithm in Section V-B 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 non-shared 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 III). Our approach can handle internal events if the following two conditions hold: (1) 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. (2) The definition of shared transitions in the merge algorithm in Section V-B 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 behavioural matching in Section IV-B, i.e., we can consider these values to be the behavioural matching similarities at iteration zero (this approach is implemented in [75]).

H. Practical Limitations

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

Evaluation. Our evaluation in Section VII may not be a comprehensive assessment of the effectiveness of our Match operator: Firstly, 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. Secondly, matching results can be improved by proper user guidance, which we did not measure here. More specifically, in Section VII, 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 VII-A and VIII-A 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.

X. Conclusions and Future Work

In this article, we presented an approach to matching and merging of Statechart models. Our Match operator includes heuristics that use both static and behavioural properties to match pairs of states in the input models. Our 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 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 behavioural models described in different notations, i.e., heterogeneous behavioural 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 article 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 behaviour-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. We are grateful to Thomas Smith and Gregory Bond for their help with our analysis of telecom features and the ECharts language. We 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," in *Workshop on Global Integrated Model Management (GaMMa '06) co-located with ICSE'06*, 2006.

[2] P. Bernstein, "Applying model management to classical meta data problems," in *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research*, 2003, pp. 209–220.

[3] S. Melnik, *Generic Model Management: Concepts And Algorithms*, ser. Lecture Notes in Computer Science. Springer, 2004, vol. 2967.

[4] D. Mandelin, D. Kimelman, and D. Yellin, "A Bayesian approach to diagram matching with application to architectural models." in *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 222–231.

[5] G. Spanoudakis and A. Finkelstein, "Reconciling requirements: A method for managing interference, inconsistency and conflict," *Annals of Software Engineering*, vol. 3, pp. 433–457, 1997.

[6] N. Maiden and A. Sutcliffe, "Exploiting reusable specifications through analogy," *Communications of the ACM*, vol. 35, no. 4, pp. 55–64, 1992.

[7] J. Rubin and M. Chechik, "A declarative approach for model composition," in *Workshop on Model-driven Approaches in Software Product Line Engineering co-located with SPLC'10*, 2010, to appear.

[8] A. Zisman, G. Spanoudakis, and J. Dooley, "A framework for dynamic service discovery," in *ASE*, 2008, pp. 158–167.

[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)," in *MoDELS*, 2006, pp. 215–229.

[11] S. Nejati, M. Sabetzadeh, M. Chechik, S. Uchitel, and P. Zave, "Towards compositional synthesis of evolving systems," in *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 285–296.

[12] S. Uchitel and M. Chechik, "Merging partial behavioural models," in *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004, pp. 43–52.

[13] D. Fischbein, G. Brunet, N. D'Ippolito, M. Chechik, and S. Uchitel, "Weak alphabet merging of partial behaviour models," *ACM Transactions on Software Engineering and Methodology*, 2010, to appear.

[14] A. Mehra, J. C. Grundy, and J. G. Hosking, "A generic approach to supporting diagram differencing and merging for collaborative design," in *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005, pp. 204–213.

[15] M. Sabetzadeh and S. Easterbrook, "View merging in the presence of incompleteness and inconsistency," *Requirements Engineering Journal*, vol. 11, no. 3, pp. 174–193, 2006.

[16] J. Whittle and J. Schumann, "Generating statechart designs from scenarios," in *ICSE '00: Proceedings of 22nd International Conference on Software Engineering*. ACM Press, May 2000, pp. 314–323.

[17] M. Jackson and P. Zave, "Distributed feature composition: a virtual architecture for telecommunications services," *IEEE Transactions on Software Engineering*, vol. 24, no. 10, pp. 831–847, 1998.

[18] P. Zave, "Modularity in Distributed Feature Composition," in *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," in *Proceedings of the Eighth International Symposium on Component-Based Software Engineering (CBSE)*, 2005, pp. 298–305.

[20] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave, "Matching and merging of statecharts specifications," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 54–64.

[21] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik, "Global consistency checking of distributed models with TReMer+," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 815–818.

[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 Transactions on Software Engineering*, vol. 29, no. 10, pp. 866–882, 2003.

[24] G. Bond, "An introduction to ECharts: The concise user manual," AT&T, Tech. Rep., 2008, available at: http://echarts.org/Downloads/Download-document/An-Introduction-to-ECharts-The-Concise-User-Manual-2008-05-20-v1.3-beta.html.

[25] G. Bond and H. Goguen, "ECharts: Balancing design and implementation," AT&T, Tech. Rep., 2002, available at: http://echarts.org.

[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," in *AAAI '04: Proceedings of Association for the Advancement of Artificial Intelligence*, 2004, pp. 1024–1025.

[28] S. Patwardhan and T. Pedersen, "Using WordNet-based context vectors to estimate the semantic relatedness of concepts," in *EACL 2006 Workshop on Making Sense of Sense – Bringing Computational Linguistics and Psycholinguistics Together*, 2006, pp. 1–8.

[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*, 2nd ed. The MIT Press, 2001.

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

[32] R. D. Nicola, U. Montanari, and F. Vaandrager, "Back and forth bisimulations," in *CONCUR '90: Proceedings of 8th International Conference on Concurrency Theory*, ser. Lecture Notes in Computer Science, vol. 458, 1990, pp. 152–165.

[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*, 1st ed. Addison Wesley, 2004.

[35] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik, "Consistency checking of conceptual models via model merging," in *RE '07: Proceedings of 15th IEEE International Requirements Engineering Conference*, 2007, pp. 221–230.

[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," in *RE '03: Proceedings of the 11th IEEE International Symposium on Requirements Engineering*, 2003, pp. 138–147.

[38] S. Nejati, "Behavioural model fusion," Ph.D. dissertation, University of Toronto, 2008.

[39] A. Hussain and M. Huth, "On model checking multiple hybrid views," in *Proceedings of 1st International Symposium on Leveraging Applications of Formal Methods*, 2004, pp. 235–242.

[40] K. Larsen and B. Thomsen, "A modal process logic," in *LICS '88: Proceedings of 3rd Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1988, pp. 203–210.

[41] M. Huth, R. Jagadeesan, and D. A. Schmidt, "Modal transition systems: A foundation for three-valued program analysis," in *ESOP '01: Proceedings of 10th European Symposium on Programming*, ser. Lecture Notes in Computer Science 2028. Springer, 2001, pp. 155–169.

[42] M. Sabetzadeh, "Merging and consistency checking of distributed models," Ph.D. dissertation, University of Toronto, 2008.

[43] H. Liang, Z. Diskin, J. Dingel, and E. Posse, "A general approach for scenario integration," in *MoDELS*, 2008, pp. 204–218.

[44] K. Larsen, B. Steffen, and C. Weise, "A constraint oriented proof methodology based on modal transition systems," in *TACAS '95: Proceedings of First International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 1019. Springer, 1995, pp. 17–40.

[45] M. Huth and S. Pradhan, "Model-checking view-based partial specifications," *Electronic Notes Theoretical Computer Science*, vol. 45, 2001.

[46] S. Nejati and M. Chechik, "Behavioural model fusion: An overview of challenges," in *ICSE Workshop on Modeling in Software Engineering (MiSE '08)*, 2008.

[47] B. Nuseibeh, J. Kramer, and A. Finkelstein, "A framework for expressing the relationships between multiple views in requirements specification," *IEEE Transaction on Software Engineering*, vol. 20, no. 10, pp. 760–773, 1994.

[48] J. Hay and J. Atlee, "Composing features and resolving interactions," in *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2000, pp. 110–119.

[49] A. Moreira, A. Rashid, and J. Araújo, "Multi-dimensional separation of concerns in requirements engineering," in *RE '05: Proceedings of the 10th IEEE International Symposium on Requirements Engineering*, 2005, pp. 285–296.

[50] P. Tarr, H. Ossher, W. Harrison, and S. S. Jr., "N degrees of separation: Multi-dimensional separation of concerns," in *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, 1999, pp. 107–119.

[51] W. Harrison, H. Ossher, and P. Tarr, "General composition of software artifacts," in *5th International Symposium Software Composition (SC'06), co-located with ETAPS'06*, ser. Lecture Notes in Computer Science, vol. 4089. Springer, 2006, pp. 194–210.

[52] E. Rahm and P. Bernstein, "A survey of approaches to automatic schema matching," *The VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[53] K. Ryan and B. Mathews, "Matching conceptual graphs as an aid to requirements re-use," in *RE '93: Proceedings of IEEE International Symposium on Requirements Engineering*, 1993, pp. 112–120.

[54] T. Alspaugh, A. Antón, T. Barnes, and B. Mott, "An integrated scenario management strategy," in *RE '99: Proceedings of the 4th IEEE International Symposium on Requirements Engineering*, 1999, pp. 142–149.

[55] Z. Xing and E. Stroulia, "Differencing logical uml models," *Autom. Softw. Eng.*, vol. 14, no. 2, pp. 215–259, 2007.

[56] U. Kelter and M. Schmidt, "Comparing state machines," in *CVSM'08: ICSE'08 Workshop on Comparison and versioning of software models*, 2008, pp. 1–6.

[57] N. Lohmann, "Correcting deadlocking service choreographies using a simulation-based graph edit distance," in *BPM '08: Proceedings of the 6th International Conference on Business Process Management*, 2008, pp. 132–147.

[58] J. Quante and R. Koschke, "Dynamic protocol recovery," in *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, pp. 219–228.

[59] K. Bogdanov and N. Walkinshaw, “Computing the structural difference between state-based models,” in *WCRE ’09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, 2009, pp. 177–186.
[60] F. van Breugel, “Abe ’08: Workshop on approximate behavioural equivalences,” 2008.
[61] L. de Alfaro, M. Faella, and M. Stoelinga, “Linear and branching metrics for quantitative transition systems,” in *ICALP ’04: Proceedings of the 31st International Colloquium on Automata, Languages and Programming*, 2004, pp. 97–109.
[62] O. Sokolsky, S. Kannan, and I. Lee, “Simulation-based graph similarity,” in *TACAS ’06: Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 3920. Springer, 2006, pp. 426–440.
[63] M. Alanen and I. Porres, “Difference and union of models,” in *UML ’03: Proceedings of the 6th International Conference on The Unified Modeling Language*, ser. Lecture Notes in Computer Science, vol. 2863. Springer, 2003, pp. 2–17.
[64] K. Letkeman, “Comparing and merging UML models in IBM rational software architect,” IBM, Tech. Rep., 2006, http://www-306.ibm.com/software/awdtools/architect/swarchitect/.
[65] A. Zito, Z. Diskin, and J. Dingel, “Package merge in UML 2: Practice vs. theory?” in *MoDELS ’06: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*, 2006, pp. 185–199.
[66] A. Boronat, J. Carsí, I. Ramos, and P. Letelier, “Formal model merging applied to class diagram integration,” *Electron. Notes Theor. Comput. Sci.*, vol. 166, pp. 5–26, 2007.
[67] D. Jackson, “Alloy: a lightweight object modelling notation,” *ACM Transactions on Software Engineering 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,” in *BDA*, 2005.
[69] S. Horwitz, J. Prins, and T. Reps, “Integrating noninterfering versions of programs,” *ACM Transaction on 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,” in *RE ’01: Proceedings of the 6th IEEE International Symposium on Requirements Engineering*, 2001, pp. 48–55.
[72] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw, “SCR*: A toolset for specifying and analyzing requirements,” in *Annual Conf. on Computer Assurance*, 1995, pp. 109–122.
[73] S. Easterbrook and M. Chechik, “A framework for multi-valued reasoning over inconsistent viewpoints,” in *ICSE ’01: Proceedings of the 23rd International Conference on Software Engineering*, 2001, pp. 411–420.
[74] J. Bayer, J. Girard, M. Würthner, J. DeBaud, and M. Apel, “Transitioning legacy assets to a product line architecture,” in *ESEC / SIGSOFT FSE*, 1999, pp. 446–463.
[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,” in *MoDELS ’07: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, 2007, pp. 514–528.
[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,” in *ETX*, 2007, pp. 55–59.
[78] R. Salay, “Using modeler intent in software engineering,” Ph.D. dissertation, University of Toronto, 2010.
[79] M. Chechik, S. Nejati, and M. Sabetzadeh, “A relationship-based approach to model integration,” *Innovations in Systems and Software Engineering*, 2011, (To Appear).
[80] A. Classen, P. Heymans, T. T. Tun, and B. Nuseibeh, “Towards safer composition,” in *ICSE Companion*, 2009, pp. 227–230.
[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,” in *ICALP ’99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, vol. 1644. Springer, 1999, pp. 169–178.
[83] D. Dams, R. Gerth, and O. Grumberg, “Abstract interpretation of reactive systems,” *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 19, pp. 253–291, 1997.

Shiva Nejati is a research scientist at the Simula Research Laboratory in Norway. She received her B.Sc. from Sharif University of Technology (Iran) in 2000, and her M.Sc. and Ph.D. from the University of Toronto in 2003 and 2008, respectively. Her main research area is software engineering, with specific interests in model-based development, behaviour analysis, requirements engineering, specification and design methods, and web-services.

Mehrdad Sabetzadeh is a Research Scientist at the Simula Research Laboratory. He received his Ph.D. from the University of Toronto in 2008 and worked as a Postdoctoral Researcher at University College London in 2009. Dr. Sabetzadeh’s main research interest is model-based software development with an emphasis on requirements engineering, verification and validation, and certification. Sabetzadeh is a Member of the IEEE Computer Society.

Marsha Chechik is currently Professor and Vice Chair in the Department of Computer Science at the University of Toronto. She received her Ph.D. from the University of Maryland in 1996. Prof. Chechik’s research interests are in the application of formal methods to improve the quality of software. She has authored over 90 papers in formal methods, software specification and verification, computer security and requirements engineering. In 2002-2003, Prof. Chechik was a visiting scientist at Lucent Technologies in Murray Hill, NY and at Imperial College, London UK. She is an associate editor of 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 Journal on Software and Systems Modeling. She regularly serves on program committees of international conferences in the areas of software engineering and automated verification. Marsha Chechik was a Co-Chair of the 2008 International Conference on Concurrency Theory (CONCUR), Program Committee Co-Chair of the 2008 International Conference on Computer Science and Software Engineering (CASCON), and Program Committee Co-Chair of the 2009 International Conference on Formal Aspects of Software Engineering (FASE). She is a Member of the IEEE Computer Society.

Steve Easterbrook is a professor of computer science at the University of Toronto. He received his Ph.D. (1991) in Computing from Imperial College in London (UK), 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 modelling and analysis of complex software 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. Easterbrook is a Member of the IEEE Computer Society.

Pamela Zave received an A.B. degree in English from Cornell University, and a Ph.D. degree in computer sciences from the University of Wisconsin–Madison. She has been with AT&T Research (formerly Bell Labs) since 1981. She is interested in all aspects of formal methods for software engineering as applied to networks, and holds 17 patents in the telecommunication area.

Dr. Zave is an ACM Fellow and an AT&T Fellow. She has received three Ten-Year Most Influential Paper awards and four Best Paper awards. She is currently chair of IFIP Working Group 2.3 on Programming Methodology.

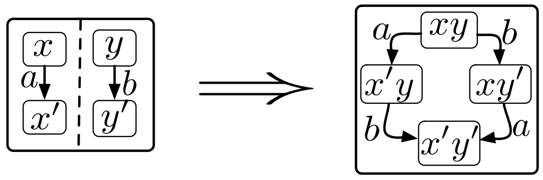


Fig. 20. Resolving AND-states (parallel states) in ECharts.

XI. Appendix

A. Flattening Statechart Models

Flattening is known as the process of removing hierarchy in Statechart models. Our merge procedure, described in Section V, is defined over hierarchical state machines, and hence, no flattening is required prior to its application. The *semantics* of our merge procedure, however, is defined over flattened Statechart models, i.e., Labelled Transition Systems (LTSs) [31] (see Definition 3), and therefore, we need to formally describe how hierarchical Statechart models are converted to flat state machines.

To do so, we first need to remove all parallelism from Statechart models. ECharts use parallel states with interleaved transition executions [24] which can be translated to the above formalism using the interleaving semantics of [23]. A simple example of this translation is shown in Figure 20. Note that more than one transition may be enabled in a parallel Statechart model. ECharts provide three kinds of priority rules to let the modellers control which transition(s) will fire when multiple ones are enabled: *Message Class Rule*, *Source Coverage Rule*, and *Transition Depth Rule* (see Section 4 of [24] for a full description of these rules). If none of these rules apply in a given situation, the transition(s) to fire are chosen non-deterministically.

We translate the resulting hierarchical Statechart without parallelism into an intermediate state machine formalism given in Definition 4, and then discuss how this formalism can be converted to LTSs.

Definition 3 (LTS): [31] An LTS is a tuple (S, s_0, R, E) where S is a set of states, $s_0 \in S$ is an initial state, $R \subseteq S \times E \times S$ is a set of transitions, and E is a set of actions.

An example LTS is shown in Figure 5(b). A *trace* of an LTS L is a *finite* sequence σ of actions that L can perform starting at its initial state. For example, ϵ, a, $a \cdot c$, and $a \cdot c \cdot c$ are examples of traces of the LTS in Figure 5(b). The set of all traces of L is called the language of L, denoted $\mathcal{L}(L)$. Let Σ be a set of symbols. We say $\sigma = e_0 e_1 \ldots e_n$ is a trace over Σ if $e_i \in \Sigma$ for every $0 \le i \le n$. We denote by Σ^* the set of all finite traces over Σ.

Let L be an LTS, and $E' \subseteq E$. We define $L@E'$ to be the result of restricting the set of actions of L to E', i.e., replacing actions in $E \setminus E'$ with the unobservable action τ and reducing E to E'. For an LTS L with τ-labelled transitions, we consider $\mathcal{L}(L)$ to be the set of traces of L with the occurrences of τ removed.

Definition 4 (State Machine): A *state machine* is a tuple $SM = (S, s_0, R, E, Act)$, where S is a finite set of states, $s_0 \in S$ is the initial state, $R \subseteq S \times E \times Act \times S$ is a transition relation, E is a set of input events, and Act is a set of output actions.

State machines in Definition 4 are similar to LTSs except that state machine transitions are labelled by (e, α), where e is an input event and α is a sequence of output actions. In contrast, LTS transitions are labelled with single actions. State machines can be translated to LTSs by replacing each transition labelled with (e, α) by a sequence of transitions labelled with single actions of the sequence $e \cdot \alpha$. In the rest of this appendix, we assume that the result of Statechart flattening is an LTS. That is, we assume that state machines are replaced by their equivalent LTSs. Note that in LTSs, we keep input events E and output actions Act distinct. So, for example, if label a appears in $E \cap Act$ of a state machine M, we keep two distinct copies of a (one for input and one for output) in the vocabulary of the corresponding LTS.

Definition 5 (Flattening): Let $M = (S, \hat{s}, <_h, E, V, R)$ be a Statechart model. For any state $s \in S$, let $Parent(s)$ be the set of ancestors of s (including s) with respect to the hierarchy tree $<_h$. We define a state machine $SM_M = (S', s'_0, R', E', Act')$ corresponding to M as follows:

$$\begin{aligned}
S' &= \{s \mid s \in S \wedge \text{s is a leaf with respect to } <_h\} \\
s'_0 &= \{s \mid s \in \hat{s} \wedge \text{s is a leaf with respect to } <_h\} \\
R' &= \{(s, e, \alpha, s') \mid \exists s_1 \in Parent(s) \cdot \exists s_2 \in Parent(s') \cdot \\
&\qquad \langle s_1, e', c, \alpha, s_2, prty\rangle \in R \wedge e = e'[c] \wedge \\
&\qquad \text{the value of } ptry \text{ is higher than other} \\
&\qquad \text{outgoing transitions of } s \text{ (and of ancestors} \\
&\qquad \text{of } s\text{) enabled by event } e \text{ and guard } c\} \\
E' &= \{e \mid \exists \langle s, e', c, \alpha, s'\rangle \in R \cdot e = e'[c]\} \\
Act' &= \{a \mid \exists \langle s, e', c, \alpha, s'\rangle \in R \cdot a \text{ appears in the sequence } \alpha\}
\end{aligned}$$

Informally, to flatten a hierarchical state machine M: (1) We keep only the leaf states of M (with respect to $<_h$). All super-states are removed. (2) We push the outgoing and incoming transitions of the super-states (non-leaf states) down to their leaf sub-states. Any incoming (resp. outgoing) transition of a super-state s is replaced by incoming (resp. outgoing) transitions to every leaf sub-state of s. (3) When a leaf state has several outgoing transitions with the same triggering event, we keep the transition with the highest priority. (4) We assume that the guards are part of the event labels and remove the set of variables of M. For example, the LTS in Figure 21(b) is the flattened form of the Statechart model in Figure 21(a). Similarly, LTSs corresponding to the Statechart models in Figure 1 are shown in Figure 22. The LTS corresponding to the Statechart model in Figure 5(a) is shown in Figure 5(b). It illustrates how we resolve priorities during flattening.

Obviously, flattening increases the number of transitions. In situations where superstates share the same sub-states (see Figure 21(c) for an example), flattening also increases the number of states because multiple copies of sub-states are created in the flattened state machine. However, since we use LTSs only to define the semantics of merge, the size increase is not a limitation in our work. For an efficient technique for flattening hierarchical state machines with super-states sharing the same substates, see [82].

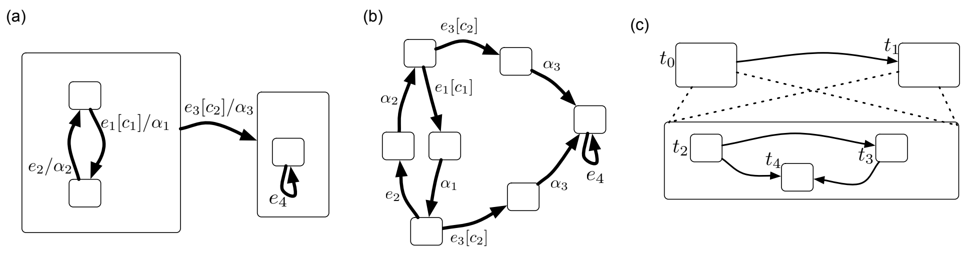


Fig. 21. Statecharts flattening: (a) An example Statecharts, (b) flattened state machine equivalent to the Statecharts in (a), and (c) an example Statecharts whose super-states share the same sub-states.

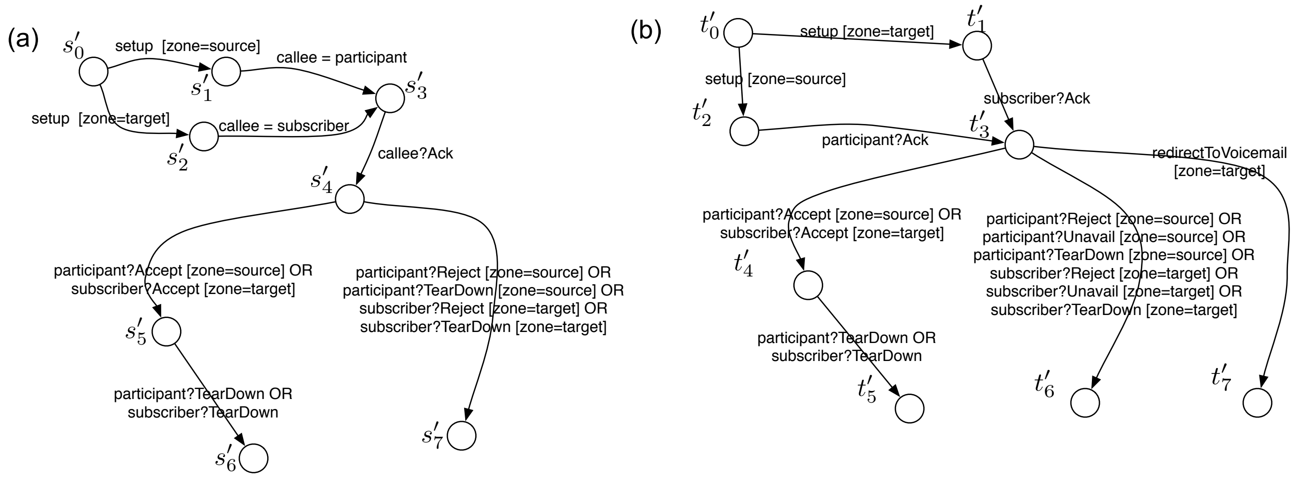

(a)
s'_0
setup [zone=source]
s'_1
callee = participant
s'_3
setup [zone=target]
callee = subscriber
s'_2
callee?Ack
s'_4
participant?Accept [zone=source] OR
subscriber?Accept [zone=target]
participant?Reject [zone=source] OR
participant?TearDown [zone=source] OR
subscriber?Reject [zone=target] OR
subscriber?TearDown [zone=target]
s'_5
participant?TearDown OR
subscriber?TearDown
s'_7
s'_6
(b)
t'_0
setup [zone=target]
t'_1
setup [zone=source]
subscriber?Ack
t'_2
t'_3
participant?Ack
redirectToVoicemail
[zone=target]
participant?Accept [zone=source] OR
subscriber?Accept [zone=target]
participant?Reject [zone=source] OR
participant?Unavail [zone=source] OR
participant?TearDown [zone=source] OR
subscriber?Reject [zone=target] OR
subscriber?Unavail [zone=target] OR
subscriber?TearDown [zone=target]
t'_4
participant?TearDown OR
subscriber?TearDown
t'_5
t'_6
t'_7

Fig. 22. LTSs generated by flattening the Statechart models in Figure 1.

B. Mixed LTSs

Individual models of variant features such as those shown in Figure 22 can be described as LTSs; however, their merge cannot. This is because LTSs do not provide any means to distinguish between different kinds of system behaviours. In particular, in our work, we need to distinguish between behaviours that are common among different variants, and behaviours about which variants disagree. In product line engineering, the former type of behaviours is referred to as *commonalities*, and the latter – as *variabilities* [34]. To specify behavioural commonalities and variabilities, we extend LTSs to have two types of transitions: One representing shared behaviours (to capture commonalities) and the other representing non-shared behaviours (to capture variabilities).

Definition 6 (Mixed LTSs): A *Mixed LTS* is a tuple $L = (S, s_0, R^{shared}, R^{nonshared}, E)$, where $L^{shared} = (S, s_0, R^{shared}, E)$ is an LTS representing *shared* behaviours, and $L^{nonshared} = (S, s_0, R^{nonshared}, E)$ is an LTS representing *non-shared* behaviours. We denote the set of shared and non-shared transitions of a Mixed LTS by $R^{all} = R^{shared} \cup R^{nonshared}$, and the LTS, (S, s_0, R^{all}, E), by L^{all}.

Every LTS (S, s_0, R, E) can be viewed as a Mixed LTS whose set of non-shared transitions is empty, i.e., $(S, s_0, R, \emptyset, E)$. Our notion of Mixed LTS is inspired by that of MixTS [83]. Yet, while both types of systems have different transition types, in MixTSs they are used to explicitly model possible and required behaviours of a system, whereas in Mixed LTSs they differentiate between shared and non-shared behaviours in variant features.

We define a notion of refinement to formalize the relationship between Mixed LTSs based on the degree of behavioural variabilities they can capture. For states s and s' of an LTS L, we write $s \overset{\tau}{\Longrightarrow} s'$ to denote $s(\overset{\tau}{\longrightarrow})^* s'$. For $e \neq \tau$, we write $s \overset{e}{\Longrightarrow} s'$ to denote $s(\overset{\tau}{\Longrightarrow})(\overset{e}{\longrightarrow})(\overset{\tau}{\Longrightarrow})s'$. For states s and s' of a Mixed LTS L, we write $s \overset{e}{\Longrightarrow}^{shared} s'$ to denote $s \overset{e}{\Longrightarrow} s'$ in L^{shared}, $s \overset{e}{\Longrightarrow}^{nonshared} s'$ to denote $s \overset{e}{\Longrightarrow} s'$ in $L^{nonshared}$, and $s \overset{e}{\Longrightarrow}^{all} s'$ to denote $s \overset{e}{\Longrightarrow} s'$ in L^{all}.

Definition 7 (Refinement): Let L_1 and L_2 be Mixed LTSs such that $E_1 \subseteq E_2$. A relation $\rho \subseteq S_1 \times S_2$ is a *refinement*, where $\rho(s,t)$ iff

1) $\forall s' \in S_1 \cdot \forall e \in E_1 \cup \{\tau\} \cdot s \overset{e}{\longrightarrow}^{all} s' \Rightarrow \exists t' \in S_2 \cdot t \overset{e}{\Longrightarrow}^{all} t' \wedge \rho(s', t')$
2) $\forall t' \in S_2 \cdot \forall e \in E_2 \cup \{\tau\} \cdot t \overset{e}{\longrightarrow}^{shared} t' \Rightarrow \exists s' \in S_1 \cdot s \overset{e}{\Longrightarrow}^{shared} s' \wedge \rho(s', t')$

We say that L_2 refines L_1, written $L_1 \preceq L_2$, if there is a refinement ρ such that $\rho(s_0, t_0)$, where s_0 and t_0 are the initial states of L_1 and L_2, respectively.

Intuitively, refinement over Mixed LTSs allows one to convert shared behaviours into non-shared ones, while preserving all of the already identified non-shared behaviours. More specifically, if L_2 refines L_1, then *every* behaviour of L_1

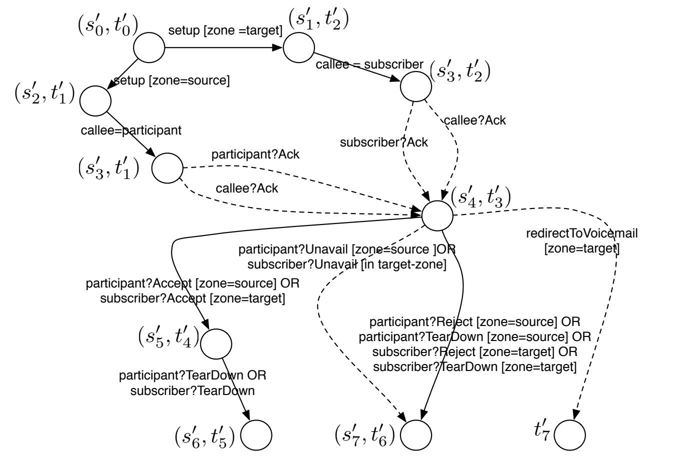

(s'_0, t'_0)
setup [zone =target]
(s'_1, t'_2)
setup [zone=source]
callee = subscriber
(s'_2, t'_1)
(s'_3, t'_2)
callee=participant
callee?Ack
subscriber?Ack
(s'_3, t'_1)
participant?Ack
callee?Ack
(s'_4, t'_3)
redirectToVoicemail [zone=target]
participant?Unavail [zone=source]OR subscriber?Unavail [in target-zone]
participant?Accept [zone=source] OR subscriber?Accept [zone=target]
participant?Reject [zone=source] OR participant?TearDown [zone=source] OR subscriber?Reject [zone=target] OR subscriber?TearDown [zone=target]
(s'_5, t'_4)
participant?TearDown OR subscriber?TearDown
(s'_6, t'_5)
(s'_7, t'_6)
t'_7

Fig. 23. Mixed LTS generated by flattening the Statechart models in Figure 10: Shared transitions are shown as solid arrows, and non-shared transitions as dashed arrows.

is present in L_2 either as shared or as non-shared: Shared behaviours of L_1 may turn into non-shared behaviours, but its non-shared behaviours are preserved in L_2. Dually, L_2 may have some additional non-shared behaviours, but all of its shared behaviours are present in L_1. As indicated in Definition 7, the vocabulary of L_1 is a subset of that of L_2. This is because the non-shared transitions of L_2 may not necessarily be present in L_1, and hence, they can be labelled by actions in $L_2 \setminus L_1$. Our notion of refinement is very similar to that given in [83] over MixTSs. The difference is that the refinement in [83] captures the "more defined than" relation between two partial models, whereas in our case, a model is more refined if it can capture more behavioural variability.

Figure 23 shows a Mixed LTS where shared transitions are shown as solid arrows, and non-shared transitions – as dashed arrows. Mixed LTS in Figure 23 refines the LTS in Figure 22(a) with the following refinement relation:

$$\{(s,(s,x)) \quad | \quad s \text{ and } (s,x) \text{ are states in Figures 22(a) and 23, respectively.}\}$$

Theorem 2: Let L_1 and L_2 be Mixed LTSs where $L_1 \preceq L_2$. Then,

1) $\mathcal{L}(L_1^{all}) \subseteq \mathcal{L}(L_2^{all})$
2) $\mathcal{L}(L_2^{shared}) \subseteq \mathcal{L}(L_1^{shared})$

Proof: By Definition 6, every Mixed LTS L has fragments L^{shared} and L^{all} which are expressible as LTSs. By Definition 7 and definition of simulation over LTSs in [31], for two Mixed LTSs L_1 and L_2 such that L_2 refines L_1, we have that L_2^{all} simulates L_1^{all}, and L_1^{shared} simulates L_2^{shared}. Based on the property of simulation [31], we have $\mathcal{L}(L_1^{all}) \subseteq \mathcal{L}(L_2^{all})$ and $\mathcal{L}(L_2^{shared}) \subseteq \mathcal{L}(L_1^{shared})$. ■

For example, consider the model in Figure 22(a) and its refinement in Figure 23. The model in Figure 22(a) is an LTS, and hence, its set of non-shared traces is empty. Every trace in this model is present in the model in Figure 23 either as a shared or a non-shared trace, i.e., $\mathcal{L}(L_1^{all}) \subseteq \mathcal{L}(L_2^{all})$. Also, every shared trace in the model in Figure 23 is present in Figure 22(a), i.e., $\mathcal{L}(L_2^{shared}) \subseteq \mathcal{L}(L_1^{shared})$. Finally, the model in Figure 23 has some non-shared traces that are not present in the model in Figure 22(a), e.g., the trace generated by the path $(s'_0, t'_0) \rightarrow (s'_1, t'_2) \rightarrow (s'_3, t'_2) \rightarrow (s'_4, t'_3) \rightarrow t'_7$. This shows that $\mathcal{L}(L_2^{nonshared})$ is not necessarily a subset of $\mathcal{L}(L_1^{nonshared})$.

C. Proof of Correctness for Merging Statechart models

Given input Statechart models M_1 and M_2, and their merge $M_1 +_\rho M_2$, let L_1 and L_2 be the LTSs corresponding to M_1 and M_2, respectively, and let L_{1+2} be the Mixed LTS corresponding to $M_1 +_\rho M_2$. We show that L_{1+2} is a common refinement of L_1 and L_2, i.e., L_{1+2} refines both L_1 and L_2.

Theorem 3: Let M_1, M_2, $M_1 +_\rho M_2$ be given, and let L_1, L_2, and L_{1+2} be their corresponding flat state machines, respectively. Let Act_1 and Act_2 be the set of output actions of M_1 and M_2, respectively, and let E_1 and E_2 be the set of labels of L_1 and L_2, respectively. Then, $L_1 \preceq L_{1+2}@\{E_1 \uplus E_2 \setminus Act_2\}$ and $L_2 \preceq L_{1+2}@\{E_1 \uplus E_2 \setminus Act_1\}$, where $\uplus$ denotes the disjoint union operator over sets.

Before we give the proof, we provide an inductive definition, equivalent to Definition 7, for the refinement relation $\preceq$.

Definition 8: We define a sequence of refinement relations $\preceq^0$, $\preceq^1$, ... on $S_1 \times S_2$ as follows:

- $\preceq^0 = S_1 \times S_2$
- $s \preceq^{n+1} t$ iff

$$\begin{array}{l}\forall s' \in S_1 \cdot \forall e \in E_1 \cup \{\tau\} \cdot s \xrightarrow{e}{}^{all} s' \Rightarrow \\ \exists t' \in S_2 \cdot t \overset{e}{\Longrightarrow}{}^{all} t' \wedge s' \preceq^n t' \\ \forall t' \in S_2 \cdot \forall e \in E_2 \cup \{\tau\} \cdot t \xrightarrow{e}{}^{shared} t' \Rightarrow \\ \exists s' \in S_1 \cdot s \overset{e}{\Longrightarrow}{}^{shared} s' \wedge s' \preceq^n t'\end{array}$$

The largest refinement relation is defined as $\bigcap_{i \geq 0} \preceq^i$. Note that since L_1 and L_{1+2} are finite structures, the sequence $\preceq^0$, $\preceq^1$, ... is finite as well.

Proof: To prove $L_1 \preceq L_{1+2}@\{E_1 \uplus E_2 \setminus Act_2\}$, we show that the relation

$$\rho_1 = \{(s,s) \mid s \in S_1 \wedge s \in S_+\} \cup \{(s,(s,t)) \mid s \in S_1 \wedge (s,t) \in S_+\}$$

is a refinement relation between L_1 and L_{1+2}.

In this proof, we assume that any tuple $(s,t) \in \rho$ where s is a state in M_1 and t a state in M_2 is replaced by its corresponding tuples (s',t') such that s' is a corresponding state to s in L_1 and t' is a corresponding state to t in L_{1+2}. For example, relation ρ in Figure 8(d), which is defined between the Statechart models in Figure 1, is replaced by the relation $\{(s'_0,t'_0),(s'_1,t'_2),(s'_2,t'_1),(s'_3,t'_1),(s'_3,t'_2),(s'_4,t'_3),(s'_5,t'_4),$ $(s'_6,t'_5),(s'_7,t'_6)\}$ between the flat LTSs in Figure 22.

To show that ρ_1 is a refinement, we prove that ρ_1 is a subset of the largest refinement relation, i.e., $\rho_1 \subseteq \bigcap_{i \geq 0} \preceq^i$. The proof follows by induction on i:

Base case. $\rho_1 \subseteq \preceq^0$. Follows from the definition of ρ_1 and the fact that $\preceq^0 = S_1 \times S_+$.

Inductive case. Suppose $\rho_1 \subseteq \preceq^i$. We prove that $\rho_1 \subseteq \preceq^{i+1}$.

By Definition 8, we need to show for every $(s,r) \in \rho_1$,

1. $\forall s' \in S_1 \cdot \forall e \in E_1 \cup \{\tau\} \cdot (s \xrightarrow{e}{}^{all} s') \Rightarrow \exists r' \in S_+ \cdot (r \overset{e}{\Longrightarrow}{}^{all} r' \wedge s' \preceq^i r')$

2. $\forall r' \in S_+ \cdot \forall e \in (E_1 \uplus E_2 \cup \{\tau\}) \setminus Act_2 \cdot (r \xrightarrow{e}^{shared} r') \Rightarrow \exists s' \in S_1 \cdot (s \overset{e}{\Longrightarrow}^{shared} s' \wedge s' \preceq^i r')$

To prove **1.**, we identify four cases:

Case 1: $\forall s' \in S_1 \cdot \forall e \in E_1 \cup \{\tau\} \cdot s \xrightarrow{e}^{all} s' \wedge$
$\exists t, t' \in S_2 \cdot (s,t) \in \rho \wedge (s',t') \in \rho$
$\Rightarrow$ (by construction of $M_1 +_\rho M_2$ in Section V-B and definition of ρ_1)
$r = (s,t) \wedge \exists (s',t') \in S_+ \cdot (s,t) \xrightarrow{e}^{all} (s',t') \wedge$
$(s', (s',t')) \in \rho_1$
$\Rightarrow$ (by the inductive hypothesis, and let $r' = (s',t')$)
$\exists r' \in S_+ \cdot r \xrightarrow{e}^{all} r' \wedge s' \preceq^i r'$

Case 2: $\forall s' \in S_1 \cdot \forall e \in E_1 \cup \{\tau\} \cdot s \xrightarrow{e}^{all} s' \wedge$
$\exists t \in S_2 \cdot (s,t) \in \rho \wedge \nexists t' \in S_2 \cdot (s',t') \in \rho$
$\Rightarrow$ (by construction of $M_1 +_\rho M_2$ in Section V-B)
$r = (s,t) \wedge \nexists t' \in S_2 \cdot (s',t') \in \rho \wedge (s,t) \xrightarrow{e}^{all} s'$
$\Rightarrow$ (by definition of S_+ and ρ_1)
$\exists t \in S_2 \cdot r = (s,t) \wedge \exists s' \in S_+ \cdot (s,t) \xrightarrow{e}^{all} s' \wedge$
$(s', s') \in \rho_1$
$\Rightarrow$ (by the inductive hypothesis, and let $r' = s'$)
$\exists r' \in S_+ \cdot r \xrightarrow{e}^{all} r' \wedge s' \preceq^i r'$

Case 3: $\forall s' \in S_1 \cdot \forall e \in E_1 \cup \{\tau\} \cdot s \xrightarrow{e}^{all} s' \wedge$
$\nexists t \in S_2 \cdot (s,t) \in \rho \wedge \exists t' \in S_2 \cdot (s',t') \in \rho$
$\Rightarrow$ (by construction of $M_1 +_\rho M_2$ in Section V-B)
$r = s \wedge \exists t' \in S_2 \cdot (s',t') \in \rho \wedge s \xrightarrow{e}^{all} (s',t')$
$\Rightarrow$ (by definition of S_+ and ρ_1)
$\exists (s',t') \in S_+ \cdot s \xrightarrow{e}^{all} (s',t') \wedge (s', (s',t')) \in \rho_1$
$\Rightarrow$ (by the inductive hypothesis, and let $r' = (s',t')$)
$\exists r' \in S_+ \cdot r \xrightarrow{e}^{all} r' \wedge s' \preceq^i (s',t')$

Case 4: $\forall s' \in S_1 \cdot \forall e \in E_1 \cup \{\tau\} \cdot s \xrightarrow{e}^{all} s' \wedge$
$\nexists t \in S_2 \cdot (s,t) \in \rho \wedge \nexists t' \in S_2 \cdot (s',t') \in \rho$
$\Rightarrow$ (by construction of $M_1 +_\rho M_2$ in Section V-B)
$r = s \wedge \nexists t' \in S_2 \cdot (s',t') \in \rho \wedge s \xrightarrow{e}^{all} s'$
$\Rightarrow$ (by definition of S_+ and ρ_1)
$\exists s' \in S_+ \cdot s \xrightarrow{e}^{all} s' \wedge (s', s') \in \rho_1$
$\Rightarrow$ (by the inductive hypothesis, and let $r' = s'$)
$\exists r' \in S_+ \cdot r \xrightarrow{e}^{all} r' \wedge s' \preceq^i r'$

To prove **2.**, by construction of merge in Section V-B, for any shared transition $r \xrightarrow{e}^{shared} r'$ in L_{1+2}, we have

- if $e \neq \tau$, then $\exists (s,t), (s',t') \in \rho \cdot r = (s,t) \wedge r' = (s',t')$.
- if $e = \tau$, then $\exists (s,t), (s,t') \in \rho \cdot r = (s,t) \wedge r' = (s,t')$.

$\forall r' \in S_+ \cdot \forall e \in (E_1 \uplus E_2) \setminus Act_2 \cdot r \xrightarrow{e}^{shared} r' \wedge$
$\big(\exists (s,t), (s',t') \in \rho \cdot r = (s,t) \wedge r' = (s',t') \wedge e \neq \tau \bigvee$
$\exists (s,t), (s,t') \in \rho \cdot r = (s,t) \wedge r' = (s,t') \wedge e = \tau\big)$
$\Rightarrow$ (by construction of $M_1 +_\rho M_2$ in Section V-B)
$\exists (s,t), (s',t') \in \rho \cdot r = (s,t) \wedge r' = (s',t') \wedge$
$s \xrightarrow{e}^{shared} s' \wedge e \neq \tau \bigvee$
$\exists (s,t), (s,t') \in \rho \cdot r = (s,t) \wedge r' = (s,t') \wedge$
$s \overset{e}{\Longrightarrow}^{shared} s \wedge e = \tau$
$\Rightarrow$ (by definition of ρ and ρ_1)
$\exists s' \in S_1 \cdot s \xrightarrow{e}^{shared} s' \wedge (s', (s',t')) \in \rho_1 \wedge e \neq \tau \bigvee$
$s \overset{\tau}{\Longrightarrow}^{shared} s \wedge (s, (s,t')) \in \rho_1$
$\Rightarrow$ (by the inductive hypothesis)
$\exists s' \in S_1 \cdot s \overset{e}{\Longrightarrow}^{shared} s' \wedge s' \preceq^i r'$

The above proves that $\rho_1 \subseteq \bigcap_{i \geq 0} \preceq^i$. Since ρ_1 also relates s_0 (the initial state of L_1) to (s_0, t_0) (the initial state of L_{1+2}), ρ_1 is indeed a refinement relation between L_1 and L_{1+2}. ρ_1 might not be the largest refinement relation, but any refinement relation that includes the initial states of its underlying models can preserve their temporal properties.

To prove $L_2 \preceq L_{1+2}$, we show that the relation

$$\sigma_2 = \{(t,t) \mid t \in S_+ \wedge t \in S_2\} \cup \{(t,(s,t)) \mid t \in S_2 \wedge (s,t) \in S_+\}$$

is a refinement relation between L_2 and L_{1+2}. The proof is symmetric to the one above. ■

Recall that by our definition in Section V-B, shared transitions r and r' must have identical events, conditions, and priorities, but they may generate different output actions. The reason that we do not require actions of shared transitions to be identical is that by our assumption in Section III, the input Statechart models are non-interacting, and hence, actions in one input model do not trigger any event in the other model. In our merge procedure, for any pair of shared transitions r and r', we create a single transition r'' in the merge that can produce the *union* of the actions of r and r'. Thus, the trace generated by r'' may not exactly match the traces of r and r'. For example, consider the following shared transitions in Figure 1:

$$s_2 \xrightarrow{\texttt{setup[zone=target]/callee==subscriber}} s_3 \quad \text{and}$$
$$t_1 \xrightarrow{\texttt{setup[zone=target]}} t_3$$

These transitions are lifted to the transition

$$(s_2, t_1) \xrightarrow{\texttt{setup[zone=target]/callee==subscriber}} (s_3, t_3)$$

in the merge in Figure 10, but the action callee==subscriber does not exist in Figure 1(b). Thus, we need to hide this action when comparing the merge with the model in Figure 1(b). Figure 24 shows the Mixed LTS corresponding to the merge in Figure 10 where actions callee==subscriber and callee==participant are hidden. It can be seen that this Mixed LTS is a refinement of the LTS corresponding to the model in Figure 1(b) where the refinement relation is $\{((s,t),t) \mid (s,t)$ and t are states in Figure 10 and 1(b), respectively. $\}$.

By Theorems 2 and 3, we have

(1) $\mathcal{L}(L_{1+2}^{shared} @ \{E_1 \uplus E_2 \setminus Act_2\}) \subseteq \mathcal{L}(L_1^{all})$, and $\mathcal{L}(L_{1+2}^{shared} @ \{E_1 \uplus E_2 \setminus Act_1\}) \subseteq \mathcal{L}(L_2^{all})$. That is, the set of shared, i.e., unguarded, behaviours of the merge is a subset of the behaviours of the individual input models.

(2) $\mathcal{L}(L_1^{all}) \subseteq \mathcal{L}(L_{1+2}^{all})$, and $\mathcal{L}(L_2^{all}) \subseteq \mathcal{L}(L_{1+2}^{all})$. That is, behaviours of the individual input models are present as either shared, i.e., unguarded, or non-shared, i.e., guarded, behaviours in their merge.

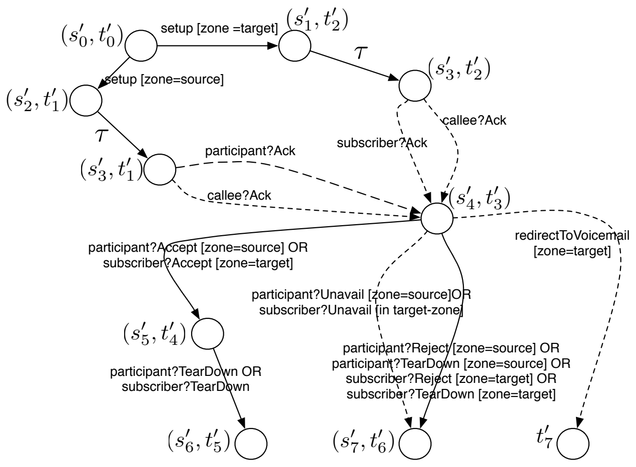

(s'_0, t'_0)
setup [zone =target]
(s'_1, t'_2)
τ
(s'_3, t'_2)
setup [zone=source]
(s'_2, t'_1)
τ
(s'_3, t'_1)
participant?Ack
callee?Ack
subscriber?Ack
callee?Ack
(s'_4, t'_3)
redirectToVoicemail [zone=target]
participant?Accept [zone=source] OR subscriber?Accept [zone=target]
participant?Unavail [zone=source]OR subscriber?Unavail [in target-zone]
(s'_5, t'_4)
participant?TearDown OR subscriber?TearDown
participant?Reject [zone=source] OR participant?TearDown [zone=source] OR subscriber?Reject [zone=target] OR subscriber?TearDown [zone=target]
(s'_6, t'_5)
(s'_7, t'_6)
t'_7

Fig. 24. Mixed LTS which is equivalent to the Statechart models in Figure 10 except that actions callee=participant and callee==subscriber are hidden.