Behavioural Model Fusion

by

Shiva Nejati

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Behavioural Model Fusion

Shiva Nejati

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2008

In large-scale model-based development, developers periodically need to combine collections of interrelated models. These models may capture different features of a system, describe alternative perspectives on a single feature, or express ways in which different features alter one another's structure or behaviour. We refer to the process of combining a set of interrelated models as *model fusion*.

A number of factors make model fusion complicated. Models may overlap, in that they refer to the same concepts, but these concepts may be presented differently in each model, and the models may contradict one another. Models may describe independent system components, but the components may interact, potentially causing undesirable side effects. Finally, models may cross-cut, modifying one another in ways that violate their syntactic or semantic properties.

In this thesis, we study three instances of the fusion problem for *behavioural models*, motivated by real-world applications. The first problem is combining *partial* models of a single feature with the goal of creating a more complete description of that feature. The second problem is maintenance of *variant* specifications of individual features. The goal here is to combine the variants while preserving their points of difference (i.e., variabilities). The third problem is analysis of interactions between models describing *different* features. Specifically, given a set of features, the goal is to construct a composition such that undesirable interactions are absent. We provide an automated tool-supported

solution to each of these problems and evaluate our solutions.

The main novelties of the techniques presented in this thesis are (1) preservation of semantics during the fusion process, and (2) applicability to large and evolving collections of models. These are made possible by explicit modelling of partiality, variability and regularity in behavioural models, and providing semantic-preserving notions for relating these models.

Contents

1 Introduction **1**

1.1 Model Fusion 1

1.2 Scope of This Thesis 4

1.2.1 Behavioural Models 5

1.2.2 Behavioural Relationships 6

1.2.3 Behavioural Fusion 8

1.2.4 Fusion Goal 10

1.3 Contributions of This Thesis 11

1.3.1 Merging Partial Feature Specifications 13

1.3.2 Merging Variant Feature Specifications 14

1.3.3 Composing Features and Analyzing Interactions 16

1.4 Organization 17

2 Preliminaries **18**

2.1 Behavioural Modelling Formalisms 18

2.1.1 State-based Formalisms 19

2.1.2 Action-based Formalisms 21

2.2 Semantics of Models 23

2.2.1 Logical Semantics of State-based Models 24

2.2.2 Trace Semantics of Action-based Models 27

2.3 Behaviour Preserving Relations 31
2.3.1 Relations over State-based Formalisms 31
2.3.2 Relations over Action-based Formalisms 33
2.4 Summary 34

3 Merging Partial Behavioural Models 35
3.1 Introduction 35
3.1.1 Motivating Example 35
3.1.2 Contributions of This Chapter 38
3.1.3 Organization of This Chapter 39
3.2 Merging Partial Consistent Models 39
3.2.1 Partial Models and Consistency 39
3.2.2 Computing Merge 44
3.3 Tool Support 53
3.4 Related Work 54
3.5 Conclusion 57

4 Merging Variant Feature Specifications 59
4.1 Introduction 59
4.1.1 Motivating Example 60
4.1.2 Contributions of This Chapter 61
4.1.3 Organization of This Chapter 62
4.2 Overview of Our Approach 63
4.3 Background 64
4.3.1 Statecharts 64
4.3.2 Flattening 66
4.3.3 Mixed LTSs 69
4.4 Assumptions 73

4.5 Matching Statecharts 74
4.5.1 Static Matching 75
4.5.2 Behavioural Matching 75
4.5.3 Combining Different Similarity Measures 78
4.5.4 Translating Similarities to Correspondences 79
4.6 Merging Statecharts 80
4.6.1 Sanity Checks for Correspondence Relations 80
4.6.2 Merge Construction 81
4.7 Tool Support 84
4.7.1 Tool Support for Matching 84
4.7.2 Tool Support for Merging 85
4.8 Evaluation 89
4.8.1 Complexity 91
4.8.2 Accuracy of Match 91
4.8.3 Correctness of Merge 94
4.9 Related Work 100
4.9.1 Matching 100
4.9.2 Merging 102
4.10 Limitations 105
4.11 Conclusion 106

5 Composing Features and Analysing Interactions 108

5.1 Introduction 108
5.1.1 Synthesizing Feature-based Systems 108
5.1.2 Contributions of This Chapter 110
5.1.3 Organization of This Chapter 112
5.2 Motivation 112
5.3 I/O Automata and Pipelines 118

5.4 Formalizing Transparency . . . 122
5.5 Compositional Synthesis . . . 129
5.6 Implementation . . . 133
5.6.1 Inputs . . . 133
5.6.2 Parallel composition . . . 134
5.6.3 Model checking . . . 134
5.6.4 Ordering permutations . . . 134
5.7 Evaluation . . . 135
5.7.1 Domain Description . . . 135
5.7.2 Experience . . . 137
5.8 Related Work . . . 141
5.8.1 Feature interaction . . . 141
5.8.2 Compositional analysis . . . 143
5.8.3 Design for verification . . . 143
5.9 Limitations . . . 144
5.10 Conclusion . . . 144

6 Conclusion **146**

6.1 Summary of The Thesis . . . 146
6.2 Future Directions for the Thesis . . . 148
6.2.1 Relationships between Models . . . 148
6.2.2 Heterogeneous Merge . . . 149
6.2.3 Tool Support . . . 149
6.2.4 Verification of Collections of Inter-related Models . . . 150
6.2.5 Software Reliability . . . 151

References **152**

A Models for the Evaluation in Chapter 4 **169**

A.1 The ECharts Language 169

A.2 Statecharts Models 170

B Models for the Evaluation in Chapter 5 **183**

B.1 The Boxtalk Language 183

B.2 Boxtalk Models 185

Index **194**

List of Tables

1.1 Three fusion problems studied in this thesis. 14

3.1 Properties of the camera models. 37

4.1 Characteristics of the studied models. 93

4.2 Tradeoff precisions and recalls. 94

5.1 Sizes of the resulting translations. 137

5.2 Negative scenarios and the resulting constraints. 139

List of Figures

1.1 Overview of the model fusion problem. 4

2.1 Examples of state-based formalisms. 20
2.2 The 3-valued Kleene logic. 21
2.3 Examples of action-based formalisms. 22
2.4 Examples of property LTSs. 29

3.1 Models of the photo-taking feature of a camera. 36
3.2 Example camera models. 40
3.3 An overview of the tool support 53
3.4 Algorithm for computing a consistency relation. 55

4.1 Simplified variants of the call logger feature. 60
4.2 Resolving AND-states. 66
4.3 Prioritizing Statecharts transitions. 67
4.4 Statecharts flattening. 69
4.5 LTSs generated by flattening the Statecharts in Figure 4.1. 70
4.6 Mixed LTS generated by flattening the Statecharts in Figure 4.9. 72
4.7 Overview of the Match operator. 74
4.8 Results of matching for call logger. 79
4.9 Resulting merge for call logger. 82
4.10 Overview of TReMer. 87

4.11 Two perspectives on a camera 88
4.12 Behavioural mapping between the models in Figure 4.11 88
4.13 A many-to-many relation between the models in Figure 4.11 89
4.14 Behavioural merge of the models in Figure 4.11 90
4.15 Results of static, behavioural, and combined matching. 90
4.16 Mixed LTS which is equivalent to the Statecharts in Figure 4.9. 99
4.17 An example showing the difference between preservation of structure and behaviour. 103

5.1 A simplified linear DFC scenario. 112
5.2 Call Blocking (CB) and Record Voice Mail (RVM). 113
5.3 Possible orderings of the features in Figure 5.2. 113
5.4 Fragments of the compositions of the features in Figure 5.2 with respect to the orderings in Figure 5.3. 114
5.5 Local ordering vs. global ordering. 117
5.6 I/O automata for the state machines in Figure 5.2. 119
5.7 Generic transparency pattern. 124
5.8 Adaptation of the generic transparency pattern to the pipeline in Figure 5.1. 124
5.9 An illustration for Theorem 5.4.1. 128
5.10 Algorithm for pipeline ordering. 130
5.11 Algorithm for finding pairwise ordering constraints. 131
5.12 Local ordering vs. global ordering. 132

A.1 Call logger vplus version. 177
A.2 Call logger voip version. 178
A.3 Merge of vplus and voip for call logger. 179
A.4 Parallel location vplus version. 180
A.5 Parallel location voip version. 181

A.6 Merge of vplus and voip for parallel location. 182

B.1 Boxtalk models: QT, NATO, AC 186

B.2 Boxtalk model: SFM 187

B.3 LTS implemented in LTSA for CB. 188

B.4 LTS implemented in LTSA for RVM. 189

B.5 LTS implemented in LTSA for QT. 190

B.6 LTS implemented in LTSA for SFM. 191

B.7 LTS implemented in LTSA for NATO. 192

B.8 LTS implemented in LTSA for AC. 193

Chapter 1

Introduction

There is a rapidly growing interest in the use of models for software engineering, driven mainly by two factors. First, there is a desire to increase the level of abstraction and automation in software development. Second, there is a need for bridging the wide conceptual gap between the problem and implementation domains. Models can capture complex systems at multiple levels of abstraction, from a variety of perspectives, allowing developers to be shielded from the complexities of the underlying implementation. The ultimate goal of model-based software development is to improve the software process by promoting the use of models as the primary artifacts of development, and to produce computer-supported technologies to transform models into running systems (France & Rumpe, 2007; Selic, 2006). Achieving this goal requires addressing a wide range of problems in model construction, management and analysis.

1.1 Model Fusion

Model-based development becomes particularly challenging in complex projects where developers have to handle a large collection of models that are inter-related. The nature of the relationships between a set of models depends primarily on the intended application of the models and how they were developed. For example, the relationships may describe

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

Several important activities in model-based development form facets of model fusion. These include:

- *Model Merging*, used to build a global view of a set of overlapping models that capture different perspectives on a certain functionality (e.g., (Sabetzadeh & Easterbrook, 2006; Nejati *et al.*, 2007; Whittle & Schumann, 2000; Uchitel & Chechik, 2004; Brunet *et al.*, 2006; Easterbrook & Chechik, 2001)). The goal of model merging is to combine the input models by unifying their overlaps. In some cases, the overlapping aspects of the input models may be conflicting. Existing merging approaches differ in handling such cases: many approaches require that only consistent models be merged, implying that inconsistent models must be repaired prior to or during merge (Letkeman, 2006). Other approaches tolerate inconsistencies by explicitly representing them in the resulting merged model (e.g. (Easterbrook & Chechik, 2001; Sabetzadeh & Easterbrook, 2006; Nejati *et al.*, 2007)). Model merging may also incorporate some kind of verification, e.g., to ensure that the results are well-formed (Sabetzadeh *et al.*, 2007b; Nentwich *et al.*, 2003).

- *Model Composition*, used to assemble a set of autonomous but interacting features that run sequentially or in parallel (e.g., (Clarke *et al.*, 1999; Hay & Atlee, 2000; Jackson & Zave, 1998; Milner, 1989)). Some of the interactions between features are desirable and intended, and some are not (Jackson & Zave, 1998). To ensure the correctness of the overall composition of a set of features, one needs to find and resolve their undesirable interactions. The techniques proposed to this end can be

expensive because the size of overall compositions grows quickly as the number of features increases. Further, the larger the number of features, the larger is the set of possible feature arrangements, and the more difficult it is to find an arrangement that does not exhibit undesirable behaviours (Nejati *et al.*, 2008). Verification of a system with a large collection of features is typically enabled by reducing reasoning about the entire system to reasoning about its individual features using assume-guarantee rules (Pnueli, 1985; Grumberg & Long, 1994; Cobleigh *et al.*, 2003) or by exploiting symmetry among the features (Emerson & Kahlon, 2000; Emerson & Namjoshi, 2003).

- *Model Weaving*, used in aspect-oriented development to incorporate cross-cutting concerns into a base system (e.g., (Moreira *et al.*, 2005; Tarr *et al.*, 1999; Harrison *et al.*, 2002; Harrison *et al.*, 2006)). Aspect-oriented approaches often come equipped with appropriate constructs for defining the type of weaving. For example, aspect-oriented programing languages provide *pointcut* constructs by which programmers specify where and when additional code, i.e., an aspect, should be executed in place of or in addition to an already defined behaviour, i.e., the main program (Kiczales *et al.*, 2001; Ossher & Tarr, 2000). In aspect-oriented modelling, weaving types are usually defined by patterns (Whittle *et al.*, 2007) to be chosen either manually or automatically using pattern matching techniques. Similar to model composition, model weaving may result in undesirable side effects. Thus, automated analysis techniques may be required to ensure that the result of weaving satisfies the desired correctness properties. In contrast to model composition, general techniques for analysis of cross-cutting aspects have not been studied much, mainly because weaving types are more diverse than the notions of parallel and sequential composition.

We note that there is a lack of consensus regarding the use of the terms “merge”, “composition” and “weaving” in the literature. For example, composition may sometimes

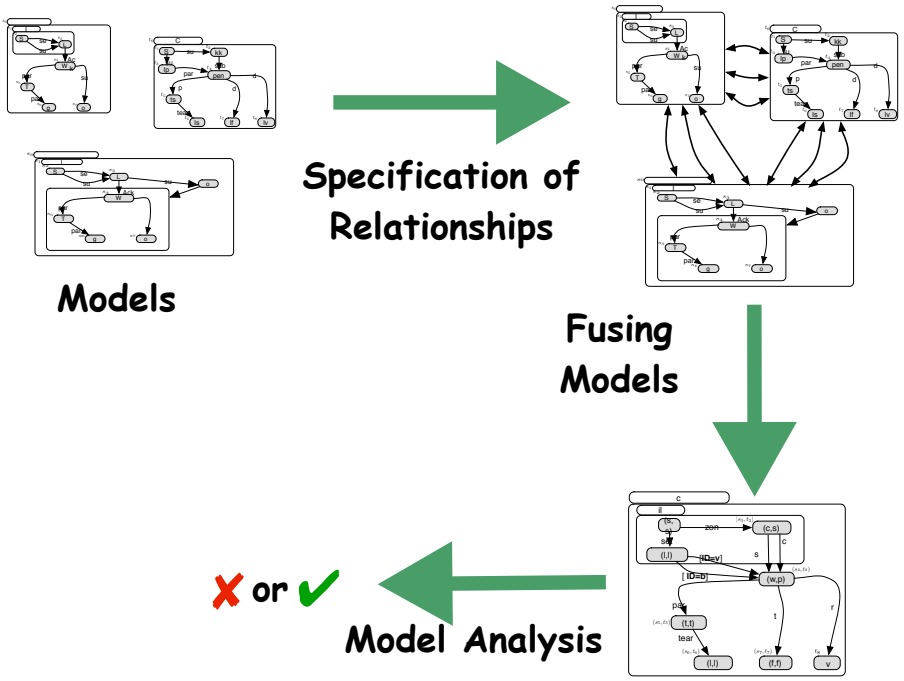


Figure 1.1: Overview of the model fusion problem.

be used to refer to what we called merging or weaving in our descriptions. To avoid confusion, in the remainder of this thesis, we follow the terminology that we established above for referring to the different fusion activities.

1.2 Scope of This Thesis

In this thesis, we study merging and composition of *behavioural models*. Behavioural models capture dynamic aspects of software systems and are described using formal notations. We use an overview of a typical model fusion problem shown in Figure 1.1 to provide a "big picture" of the research on behavioural model fusion and motivate this thesis. As shown in Figure 1.1, in every model fusion problem, we start with a set of *models* that are related. Our first task is to identify their *relationships*. Then, we need to combine these inter-related models using an appropriate *fusion activity*. At the end,

we need to analyse the result to ensure that the *goal of the fusion problem* at hand has been achieved. We identify four key abstractions in this process: *model*, *relationship*, *fusion activity*, and *fusion goal*. Providing sound engineering solutions for model fusion problems requires finding suitable implementations for these abstractions. Below, we outline notions of model, relationship, fusion activity, and fusion goal on which this thesis is grounded.

1.2.1 Behavioural Models

Behavioural modelling formalisms are broadly divided into two classes: declarative (e.g., (Jackson, 2002; Spivey, 1989)), and operational (e.g., (Milner, 1989; Clarke *et al.*, 1999; Magee & Kramer, 2006)). Declarative formalisms are highly expressive, but not easily amenable to operationalization and consistency checking. In contrast, operational models, commonly described in a state machine form (Harel & Politi, 1998; Jaffe *et al.*, 1991; Heitmeyer *et al.*, 1995), are generally less expressive, but they lend themselves to downstream development activities such as simulation, verification and code generation (Clarke *et al.*, 1996).

In this thesis, we focus on operational models described as state machines. A state machine expressed in traditional formalisms, such as LTSs (Milner, 1989) or Kripke structures (Clarke *et al.*, 1999), is assumed to be a complete description of a system up to some level of abstraction (Uchitel & Chechik, 2004). This assumption is limiting for model fusion because in this context, models may be

- *partial*, e.g., when they describe incomplete information from different perspectives (see Chapter 3).
- *inconsistent*, e.g., when they describe alternative or competing versions of an individual feature (see Chapter 4).
- *open*, e.g., when they interact with their environment, and their behaviours crucially

depend on their environmental interactions (Harel & Pnueli, 1985) (see Chapter 5).

Recent work on behavioural modelling has proposed several state machine formalisms to address the limitations of traditional notations. Of particular note are:

- Formalisms that distinguish between *required* behaviours of systems, about which the modeller has full knowledge, and *possible* behaviours, which may be elaborated incrementally in future refinements, e.g., Modal Transition Systems (Larsen & Thomsen, 1988), Partial Kripke structures (Bruns & Godefroid, 2000), Kripke Modal Transition Systems (Huth *et al.*, 2001), and Mixed Transition Systems (Dams *et al.*, 1997; Cleaveland *et al.*, 1995).

- Formalisms that distinguish between *fixed* behaviours of a system and behaviours that can be *customized* based on end-user needs, e.g., Parameterized State Machines (Gomaa, 2004) and Mixed Transition Systems (Dams *et al.*, 1997; Cleaveland *et al.*, 1995).

- Formalisms that differentiate between behaviours controlled by the *system* and those controlled by the *environment* in which the system operates, e.g., Input/Output Automata (Lynch & Tuttle, 1987), Interface Automata (de Alfaro & Henzinger, 2001), Alternating (or Agent-Based) Transition Systems (Alur *et al.*, 1997), and Tree Automata (Kupferman *et al.*, 2001).

Different fusion problems call for different modelling formalisms. For each of the problems we address in this thesis, we use a formalism that is best suited to the needs of that problem.

1.2.2 Behavioural Relationships

Relationships between models play an important role in model fusion. In model merging, relationships specify the overlaps between the input models. These relationships can be

specified either implicitly (e.g., through name equivalence if models have a common vocabulary, or identifier equivalence if models have common ancestors), or explicitly, through some notion of mapping between model elements. Explicit mappings provide the flexibility to relate models that use different vocabularies or apply a shared vocabulary inconsistently (Brunet *et al.*, 2006).

For structural models, e.g., domain and Entity Relationship diagrams, a relationship is a set of correspondences between model elements, i.e., nodes and edges. A correspondence between two elements is usually interpreted as the elements being the *same* (equal) (Sabetzadeh & Easterbrook, 2006). Under this interpretation, relating distinct elements a and b of a model M_1 to an element c of a model M_2 prescribes a collapse of all the three elements a, b, and c into a single element of the merge.

For state machines, relationships are often specified by binary relations over their state spaces. We argue that in state machine merging, equivalence is not a very useful notion for their relationships because collapsing distinct states of one model may result in the violation of certain behavioural properties (Sabetzadeh *et al.*, 2007a). To avoid this problem, we treat relationships as *behavioural similarity*, or *similarity* for short. Specifically, if two behaviourally distinct states a and b of M_1 correspond to a single state c of M_2, the merge will include distinct states (a, c) and (b, c) – the former state combines the behaviours of a and c and the latter those of b and c. We prove that a merge constructed this way can be characterized by the logical notion of *common refinement* which is provably behaviour-preserving (Hussain & Huth, 2004; Uchitel & Chechik, 2004).

In model composition, relationships describe the *communications* between independent but interacting models, by specifying the states or actions at which models synchronize their behaviours (Clarke *et al.*, 1999; Milner, 1989). For example, suppose models M and M' communicate through a channel, whereby M writes an action aliased a to the channel, and M' reads that action under alias b. The synchronization relation between M and M' is (a, b) indicating that the output action a (of M) synchronizes with

the input action b (of M'). Synchronization relations are often determined by top-level design decisions such as the overall system architecture and (pre-specified) state machine interfaces.

1.2.3 Behavioural Fusion

The focus of this thesis is on merge and composition which are key activities for combining inter-related models.

Merge. Model merging is the process of combining overlapping models with respect to their relationships. Merge is often used to combine models that are produced during a distributed development process, e.g., models built by distributed teams over time or across many geographical locations. For such models, we can never be entirely sure how the models are related and how they should be combined. 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, 2004) studies matching and merging of conceptual database schemata, (Mehra *et al.*, 2005) proposes a general framework for merging visual design diagrams, (Sabetzadeh & Easterbrook, 2006) describes an algebraic approach for merging requirements views, and (Mandelin *et al.*, 2006) provides a technique for matching architecture diagrams using machine learning. These approaches treat models as graphical artifacts while largely ignoring their semantics. This treatment provides generalizable tools that can be applied to many different modelling notations, and is particularly suited to early stages of development, when models may have loose or undefined semantics. However, a semantics-free outlook on 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 & Schumann, 2000) uses logical pre/post-conditions over object interactions for merging independently developed sequence diagrams, and (Uchitel & Chechik, 2004) uses refinement relations for merging consistent state-machine models such that their behavioural properties are preserved. These approaches, however, do not make the role of relationships explicit, and as such, do not provide explicit means for computing and manipulating relationships between models. This can make it difficult for modellers to guide the merge process, in particular, when models have discrepancies in their vocabulary and are defined at different levels of abstraction. In our work, we propose a merging process which makes identification of model relationships separate from model integration by providing two independent operators: A *Match* operator for finding model relationships, and a *Merge* operator for combining models with respect to their known relationships. We develop specific instances of these operators for hierarchical state machines in Chapter 4.

Composition. Model composition refers to the process of assembling a set of models describing autonomous but interacting features of a system, and verifying that the result is correct. In contrast to merge, composition is a well-studied notion for behavioural models: Several notions of composition have been introduced for behavioural formalisms such as LTSs and Kripke structures, and their semantic properties are well-understood (Milner, 1989; Lynch & Tuttle, 1987; Clarke *et al.*, 1999). These notions often treat models as independent components that communicate through shared interfaces, e.g., architectural links or bindings (Magee & Kramer, 2006). Existing approaches generally assume that relationships between models in a composition are known prior to the assembly phase. For example, in (Jackson & Zave, 1998; Pomakis & Atlee, 1996; Hay & Atlee, 2000), a precedence ordering derived from system requirements or specified by a domain expert determines how models are related in a linear architecture. In (Plath & Ryan, 2001; Hall,

2000), model relationships are specified through a base system, and in (Khoumsi, 1997) through a central controller.

A major problem in constructing correct compositions is finding scalable verification techniques. Verification of large compositions can be done using existing compositional verification techniques, e.g., the assume-guarantee style of reasoning (Pnueli, 1985), when the set of system components is *fixed*. However, these techniques become inadequate in systems that *evolve* over time, where components are periodically added, removed, or revised. To verify such systems, we need to design the components in a way that verification results can be reused across evolutions. To achieve this goal, we exploit behavioural design patterns that make verification *change-aware*.

1.2.4 Fusion Goal

Different fusion activities serve different purposes. In any situation where models are developed independently, merge provides a way to gain a unified perspective of the system, to explore models and their relationships, and to identify inconsistencies (or variabilities) between models (Sabetzadeh & Easterbrook, 2006; Nejati *et al.*, 2007; Sabetzadeh *et al.*, 2007b). In this thesis, we use merge for two main purposes (1) combining partial information coming from different sources, and (2) facilitating maintenance of variant system specifications by combining their commonalities and identifying their variabilities.

Composition has been mainly studied as a way to break down the construction of a large system into smaller self-contained components (Clarke *et al.*, 1999). It has also been considered for analysing interactions between different features of a system which is a pervasive problem in feature-oriented development (Jackson & Zave, 1998) and software product line engineering (Gomaa, 2004). In this thesis, we study composition of feature-based systems with the goal of ensuring that global systems do not exhibit undesirable interactions.

1.3 Contributions of This Thesis

In this thesis, we study three instances of the model fusion problem motivated by real-world applications. These problems are:

1. *Merging* partial models of a single feature with the goal of creating a more complete description of that feature.

2. *Merging* complete but variant models of an individual feature with the goal of capturing commonalities and variabilities between the variants and facilitating their maintenance.

3. *Composing* a set of features with the goal of identifying and resolving their undesirable interactions in a scalable way.

We motivate each of these problems using simple illustrative models. The general strategy to solve these problems is to formalize each problem using abstract mathematical structures, work out a theoretical solution for the problem using the research on behavioural models and logic, and turn the theoretical solution to an automated analysis tool. We evaluate our solutions by applying them to models from a telecommunication domain (Jackson & Zave, 1998) with the goal of assessing scalability and applicability of our solutions. Finally, we outline the limitations of our work by discussing (1) the practical details that we may have failed to address, and (2) directions that we have not considered in our evaluation due to the lack of proper tool support and case studies, or due to the burden of designing more realistic lab experiments.

In providing solutions to these problems, a number of significant results have been achieved, which can be summarised as follows:

Explicit formalization of partiality and variability. We show that partial state machines are appropriate formalisms for describing complementary models with different vocabulary (Chapter 3). The semantics of these state machines allows us

to explicitly specify the undefined behaviours of a model, and monotonically refine those behaviours using knowledge coming from other sources while preserving the rest of behaviours of that model.

We explicitly model behavioural variabilities between variant feature specifications using *parameterized* state machines (Chapter 4). These state machines allow us to distinguish between commonalities and variabilities of the input models in their merge. We refine a parameterized state machine by extending its parameterized (or variable) behaviours while preserving the total set of parameterized and non-parameterized behaviours. In other words, refinement of these state machines is the process of creating models that can capture a wider range of variabilities without changing the overall behaviours of the models.

Using syntactic and semantic characteristics to identify model relationships. We provide a Match operator for finding relationships between Statecharts models (Chapter 4). Our operator uses a range of heuristics including typographic and linguistic similarities between the vocabularies of different models, structure similarities between the hierarchical nesting of model elements, and semantic similarities between models based on a quantitative notion of behavioural bisimulation. We further provide an evaluation of this operator using models from a telecommunication domain, showing that our operator is effective for finding relationships between independently developed models.

Semantic preserving merge procedures. By basing our notion of merge on common refinement, we provide merge procedures that are able to preserve temporal properties, i.e., CTL/mu-calculus properties (Clarke *et al.*, 1999), and (positive/negative) traces (Grosu & Smolka, 2005), of the input models (Chapters 3 and 4). The preservation results hold for both partial state machines and state machines with variable behaviours. Furthermore, the merge procedures for state machines are im-

plemented as part of a model management tool called TReMer (Sabetzadeh *et al.*, 2007a).

Using design patterns to make synthesis/verification change-aware. We provide a change-aware algorithm for constructing system compositions that do not exhibit undesirable behaviours (Chapter 5). The formal groundwork of our approach is a behavioural design pattern that we assume to be implemented by every system component. This pattern imposes a regularity condition on the behaviour of system components, so that *local* changes, i.e., changes made to individual system components, do not have ripple effects through the system, and hence do not trigger a complete reconfiguration of the system. We use this regularity to reuse analysis results across local changes to the system, and thus making our algorithm change-aware. In this thesis, we specifically focus on feature-based systems whose components are arranged using a pipeline architectural style. We report on a prototype implementation of our algorithm, and illustrate and evaluate it by applying it to a set of AT&T telecom features.

In the remainder of this section, we briefly describe our solution to each of the three problems outlined at the beginning of Section 1.3. A comparative overview of our solutions is given in Table 1.1.

1.3.1 Merging Partial Feature Specifications

Our first study concerns combining models describing *complementary* perspectives on a single feature of a system. Since these perspectives are partial, they use only a fragment of the vocabulary that one needs in order to specify a complete description of the feature under development. Partial state machines (e.g., (Huth *et al.*, 2001)) provide an ideal formalism for describing complementary perspectives by adopting an open-world semantics. Under this semantics, predicates that are not explicitly specified to be true or false

	Merging Partial Feature Specifications	**Merging Variant Feature Specifications**	**Composing Features and Analyzing Interactions**
Models	Kripke modal transition systems	parameterized state machines	input/output state machines
Relationships	Identifying states satisfying the same temporal properties	Identifying overlapping and non-overlapping behaviours	Synthesizing feature sequences
Fusion Activity	Merging partial and consistent state machines	Merging complete and potentially inconsistent state machines	Composing independent system features
Fusion Goal	Preservation of temporal properties of the original models in their merges	Preservation of commonalities and variabilities between model variants	Constructing a composition in which undesirable behaviours are absent

Table 1.1: Three fusion problems studied in this thesis.

are treated as unknown, and left for other perspectives to determine. This is in contrast to a closed-world semantics where nothing can occur other than what is explicitly stated.

To merge partial state machines, we first check if they are *consistent*, i.e., if they agree on the set of temporal properties that they satisfy. We do so by computing a similarity relation between their states. This relation maps a pair of states s and t iff every temporal property holding in s either holds in t, or evaluates to unknown. If such a similarity relation exists, the construction of a merged model is straightforward: every pair of consistent states is merged to form a single state in the result. We show that our construction is a *common refinement* (Hussain & Huth, 2004) of the original models, and thus is behaviour-preserving.

1.3.2 Merging Variant Feature Specifications

Our second study concerns the maintenance of variant specifications of individual system features. The goal here is to merge the variants while preserving their points of difference (i.e., variabilities). In contrast to our first study, variants describe *alternative*, rather than *complementary*, descriptions of a feature. We formalize variants as parameterized state

machines (e.g., (Gomaa, 2004)). Parameterized state machines allow us to explicitly distinguish between common and variable behaviours.

In this study, variant models have major behavioural discrepancies because their development was distributed across time and over different teams of people. As a result, we cannot be entirely sure how variant models are related, making it infeasible to design an exact procedure for computing relationships between variants. Instead, we provide some assistance in identifying such relations by utilizing heuristics that imitate the reasoning of a domain expert. The relations found using such heuristics must always be reviewed by analysts and adjusted by adding any missing correspondences and removing any spurious ones. In our work, we use a number of heuristics including typographic and linguistic similarities between the vocabularies of different models, structural similarities between states, and semantic similarities between models based on a quantitative notion of behavioural bisimulation.

Having specified a relation between the variants, the merge is computed as follows: states mapped by the relation are combined and lifted to the merge; transitions with similar ending points and identical labels, i.e., shared behaviours, are lifted to the merge without any changes; and the transitions with different ending points or different labels, i.e., non-shared behaviours, are first guarded by conditions denoting their origins and then lifted to the merge. This approach guarantees that the merge preserves, in either a guarded or an unguarded form, every behaviour of the input models. As in our first study, preservation of behaviour is proven by showing that the merge is a common refinement, noting that this refinement is over parameterized state machines rather than partial state machines. We report on an implementation of our approach and evaluate it using a set of state machines describing variant specifications of telecommunication features from AT&T.

1.3.3 Composing Features and Analyzing Interactions

In our third study, we provide a solution for synthesizing *correct* feature compositions. Specifically, given a set of models describing *different* features of a system and a set of safety properties describing undesirable behaviours, we construct a composition of the models in which the given undesirable behaviours are absent. We formalize features as I/O automata, which distinguish the input, internal, and output actions of each feature. This distinction between different types of actions is crucial for properly describing communications between the features (Lynch & Tuttle, 1987).

In feature-based systems, features are typically configured in a sequential arrangement where each feature interacts with its immediate neighbours, e.g., by passing messages (Fisler & Krishnamurthi, 2005). Nevertheless, since features are distinct modules that run in parallel with one another, the overall behaviour of a system is defined as a parallel composition rather than a sequential one. While the sequential architecture of feature-based systems can potentially allow one to add or remove features dynamically, the parallel composition defies this flexibility: Adding or removing features from a parallel composition may result in unexpected undesirable behaviours. In general, when features have unrestricted designs, addition or removal of features may require the system to be reconfigured from scratch.

We identify and formalize a behavioural design pattern, called *transparency*, that is used in the feature-based development. This pattern imposes a regularity condition on the behaviour of features, so that changes made to individual features will not have ripple effects through the system, and hence will not trigger a complete reconfiguration of the system. We use this regularity to provide an efficient algorithm for synthesizing correct feature arrangements. We report on a prototype implementation of our synthesis algorithm, applying it to a set of AT&T telecom features to find a safe ordering for them in the Distributed Feature Composition (DFC) architecture (Jackson & Zave, 1998).

1.4 Organization

The rest of the thesis is organized as follows. In Chapter 2, we set our notation and introduce behavioural modelling formalisms, temporal logics, notions of composition and refinement, and model checking. In Chapter 3, we describe our technique for merging partial descriptions of a single feature (Nejati & Chechik, 2005). In Chapter 4, we describe our work on merging models of variant feature specifications (Nejati *et al.*, 2007; Sabetzadeh *et al.*, 2007a). In Chapter 5, we describe our approach to synthesizing correct compositions of feature sets (Nejati *et al.*, 2008). Finally, we conclude the thesis in Chapter 6 with a summary and an outline of our future research directions.

Chapter 2

Preliminaries

The work presented in this thesis uses different notions of behavioural modelling formalisms and behaviour preserving relations. This chapter gives an overview of these notions and fixes the notation used in the later chapters. In Section 2.1, we present an overview of behavioural models of computation. In Section 2.2, we define semantics of these models, and in Section 2.3, we review behaviour-preserving relations, and describe preservation theorems which establish connection between syntax and semantic properties of behavioural models.

2.1 Behavioural Modelling Formalisms

There are two general approaches to modelling operational behaviour of software systems: *state-based*, and *action-based*. In the state-based approach, an execution of a system is viewed as a sequence of states in which every state is an assignment of values to some set of propositions. The action-based approach views an execution as a sequence of actions. These two approaches are in principle equivalent: An action can be modeled as a state change, and a state can be modeled as an equivalence class of sequences of actions. However, the two approaches have traditionally taken very different formal directions (Abadi & Lamport, 1993). State-based approaches are often rooted in logic:

a specification can be seen as a logical formula. Action-based approaches have tended to use algebra: a specification is seen as an object that is manipulated algebraically (e.g., Milner's CCS (Milner, 1989)). In this thesis, we use both state-based and action-based formalisms.

2.1.1 State-based Formalisms

We begin by introducing a standard and widely-used state-based modelling language known as *Kripke structure*.

Definition 2.1.1 (Kripke structure) *A* Kripke structure *is a tuple* (S, s_0, R, L, AP)*, where*

- S *is a set of states;*
- $s_0 \in S$ *is an initial state;*
- $R \subseteq S \times S$ *is a transition relation;*
- AP *is a set of atomic propositions; and*
- $L : S \to 2^{AP}$ *is an interpretation function that determines what atomic propositions hold in each state.*

An example of a Kripke structure is shown in Figure 2.1(a), where

- $S = \{s_0, s_1, s_2\}$, and s_0 is the initial state;
- $R = \{(s_0, s_1), (s_0, s_2), (s_1, s_1), (s_2, s_0), (s_2, s_1)\}$
- $L(s_0) = \{p, r\}, L(s_1) = \{q, r\}, L(s_2) = \{p, q, r\}$; and
- $AP = \{p, q, r\}$

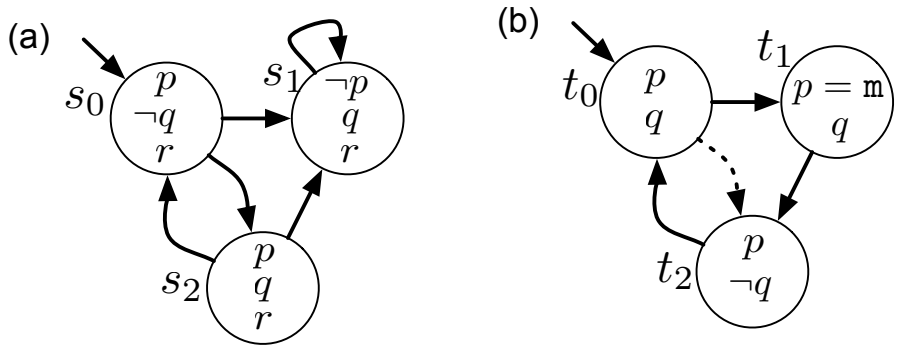


Figure 2.1: Examples of state-based formalisms: (a) A Kripke structure, and (b) a Kripke Modal Transition System (KMTS).

Kripke structures are rather primitive models of computation. We sometimes refer to Kripke structures as *classical* transition systems. More expressive state-based formalisms often extend Kripke structures with additional sets of transitions, or assume that their atomic propositions can accept non-boolean values. Below, we first introduce the 3-valued logic and then describe a non-classical state-based formalism, namely, *Kripke Modal Transition System(KMTS)* (Huth *et al.*, 2001), which is defined based on the 3-valued logic.

We denote by **3** the 3-valued Kleene logic (Kleene, 1952) with elements true (t), false (f), and maybe (m). The truth ordering $\leq$ of this logic is defined as $\mathsf{f} \leq \mathsf{m} \leq \mathsf{t}$, and negation as $\neg\mathsf{t} = \mathsf{f}$ and $\neg\mathsf{m} = \mathsf{m}$. An additional ordering $\preceq$ relates values based on the amount of information: $\mathsf{m} \preceq \mathsf{t}$ and $\mathsf{m} \preceq \mathsf{f}$, so that m represents the least amount of information. We also define the meet and the join operators with respect to $\preceq$, denoting them by $\sqcap$ and $\sqcup$, respectively. For example, $\mathsf{m} \sqcap \mathsf{t} = \mathsf{m}$. Note that $\mathsf{t} \sqcup \mathsf{f}$ is not defined. The logic **3** and its truth and information orderings are shown in Figure 2.2.

Definition 2.1.2 (KMTS) *(Huth* et al.*, 2001) A* Kripke Modal Transition System (KMTS) *is a tuple* $(S, s_0, R^{must}, R^{may}, L, AP)$*, where* S *is a set of states,* $s_0 \in S$ *is the initial state,* $R^{must} \subseteq S \times S$ *and* $R^{may} \subseteq S \times S$ *are* must *and* may *transition relations, respectively,*

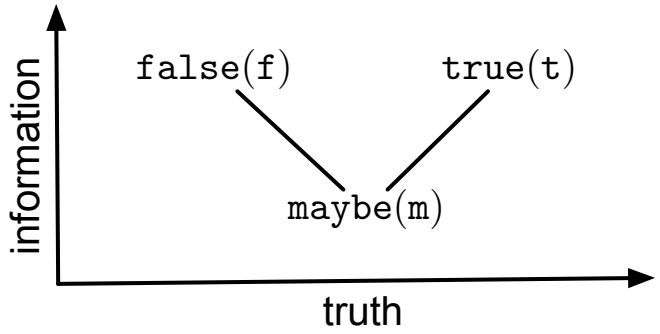


Figure 2.2: The 3-valued Kleene logic.

$L : S \to \mathbf{3}^{AP}$ is a 3-valued labelling function, and AP is the set of atomic propositions.

Using *may* transitions and 3-valued propositions, KMTSs allow explicit modelling of what is not known about the behaviour of a system. Figure 2.1(b) shows an example of a KMTS, where

- $S = \{t_0, t_1, t_2\}$ and t_0 is the initial state;
- $R^{must} = \{(t_0, t_1), (t_1, t_2), (t_2, t_0)\}$;
- $R^{may} = \{(t_0, t_1), (t_1, t_2), (t_2, t_0), (t_0, t_2)\}$;
- $L(t_0) = \{(p, \mathsf{t}), (q, \mathsf{t})\}, L(t_1) = \{(p, \mathsf{m}), (q, \mathsf{t})\}, L(t_2) = \{(p, \mathsf{t}), (q, \mathsf{f})\}$; and
- $AP = \{p, q\}$

In Figure 2.1(b), transitions that are in $R^{must} \cap R^{may}$ are shown as solid arrows, and those that are in R^{may} but not in R^{must} – as dotted arrows. Note that $R^{must} \subseteq R^{may}$ in this KMTS.

2.1.2 Action-based Formalisms

Labeled Transition System (LTS) (Milner, 1989) is a widely used action-based formalism for modelling and analyzing the behaviour of concurrent and distributed systems.

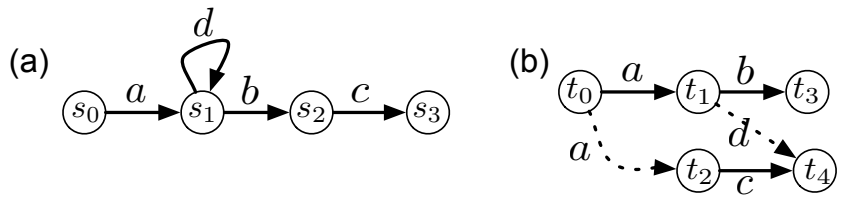


Figure 2.3: Examples of action-based formalisms: (a) An example of LTS, and (b) an example of a Mixed Transition System (MixTS).

Definition 2.1.3 (LTS) *(Milner, 1989) 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. We write a transition* $(s, e, s') \in R$ *as* $s \xrightarrow{e} s'$.

An example LTS is shown in Figure 2.3(a) where

- $S = \{s_0, s_1, s_2, s_3\}$, and s_0 is the initial state;
- $E = \{a, b, c, d\}$; and
- $R = \{(s_0, a, s_1), (s_1, b, s_2), (s_1, d, s_1)(s_2, c, s_3)\}$.

A *trace* of an LTS M is a *finite* sequence σ of actions that M can perform starting at its initial state. For example, ϵ, a, $a \cdot b$, $a \cdot b \cdot c$, and $a \cdot d \cdot d \cdot b$ are traces of the LTS in Figure 2.3(a). The set of all traces of M is called the language of M, denoted $\mathcal{L}(M)$. 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 $0 \leq i \leq n$. We denote by Σ^* the set of all finite traces over Σ.

Let M be an LTS, and $E' \subseteq E$. We define $M@E'$ to be the result of restricting the set of actions of M to E', i.e., replacing actions in $E \setminus E'$ with the unobservable action τ and reducing E to E'. For an LTS M with τ-labelled transitions, we consider $\mathcal{L}(M)$ to be the set of traces of M with the occurrences of τ removed. This is a standard way for hiding unobservable computations of LTSs (Larsen *et al.*, 1995).

Similar to Kripke structures, LTSs can be augmented with additional transition relations to express non-classical properties of systems such as partiality and inconsistency.

Below, we define *Mixed Transition Systems (MixTS)* (Larsen & Thomsen, 1988; Dams *et al.*, 1997) which is an action-based language with the same expressive power as KMTS.

Definition 2.1.4 (MixTS) *(Larsen & Thomsen, 1988; Dams* et al.*, 1997) A* Mixed Transition System (MixTS) *is a tuple* $(S, s_0, R^{must}, R^{may}, E)$ *where both* (S, s_0, R^{must}, E) *and* (S, s_0, R^{may}, E) *are LTSs.*

Figure 2.3(b) illustrates a MixTS, where

- $S = \{t_0, t_1, t_2, t_3, t_4\}$, and t_0 is the initial state;
- $E = \{a, b, c, d\}$;
- $R^{must} = \{(t_0, a, t_1), (t_1, b, t_3), (t_2, c, t_4)\}$; and
- $R^{may} = \{(t_0, a, t_1), (t_1, b, t_3), (t_2, c, t_4), (t_0, a, t_2), (t_1, d, t_4)\}$.

Similar to KMTSs, transitions that are in both R^{must} and R^{may} are shown as solid arrows in Figure 2.3(b), and those that are only in R^{may} – as dashed arrows. Given a MixTS $M = (S, s_0, R^{must}, R^{may}, E)$, we refer to $M^{must} = (S, s_0, R^{must}, E)$ and $M^{may} = (S, s_0, R^{may}, E)$ as *must* and *may* fragments of M, respectively. We write a transition $(s, e, s') \in R^{must}$ as $s \xrightarrow{e}^{must} s'$, and a transition $(s, e, s') \in R^{may}$ as $s \xrightarrow{e}^{may} s'$.

2.2 Semantics of Models

The semantics of state-based models is often described using logical specifications, in particular, temporal logics. In contrast, it is more convenient to define the semantics of action-based models as sets of traces or trees of actions that they can generate. Specifically, in this thesis, we give the semantics of action-based models using traces because they are expressive enough for the properties in our case studies in Chapters 4 and 5.

2.2.1 Logical Semantics of State-based Models

In this section, we define two temporal logics, namely, *propositional μ-calculus (L_μ)* (Kozen, 1983), and *Computational Tree Logics (CTL)* (Clarke *et al.*, 1986), and describe the semantics of Kripke structures and KMTSs using these two logics.

Temporal Logics. We begin by defining the logic L_μ.

Definition 2.2.1 (L_μ) *(Kozen, 1983) Let Var be a set of fixpoint variables, and AP be a set of atomic propositions. The* logic $L_\mu(AP)$ *is the set of formulas generated by the following grammar:*

$$\varphi ::= \mathsf{t} \mid p \mid Z \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid EX\varphi \mid \mu Z \cdot \varphi(Z)$$

where $p \in AP$, $Z \in Var$, and $\varphi(Z)$ is syntactically monotone in Z.

The derived connectives are defined as follows:

$$\begin{aligned} \varphi_1 \vee \varphi_2 &= \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ AX\varphi &= \neg EX\neg\varphi \\ \nu Z \cdot \varphi(Z) &= \neg\mu Z \cdot \neg\varphi(\neg Z) \end{aligned}$$

We often write L_μ for the set of μ-calculus formulas over some unspecified set of atomic propositions. An L_μ formula φ is *well formed* if and only if in every subformula of φ of the form $\mu Z \cdot \psi(Z)$, the fixpoint variable Z occurs under the scope of an even number of negations in ψ. From this point onwards, we consider well formed formulas only.

The intended meaning of the modal operator EX is "an existence of an immediate future". For example, if "p" means that p holds now, EXp means that there exists an immediate future where p holds, and AXp means that p holds in all immediate futures. The quantifiers μ and ν stand for the least and greatest fixpoint, respectively.

An occurrence of a variable Z in a formula φ is *bound* if it appears in the scope of a μ quantifier and is *free* otherwise. For example, Z is free in $p \vee EXZ$, and is bound in $\mu Z \cdot p \vee EXZ$. A formula φ is *closed* if it does not contain any free variables.

Definition 2.2.2 (Semantics of KMTS) *(Huth* et al.*, 2001) Let K be a KMTS, φ be an L_μ formula, and $e : Var \to \mathcal{P}(S)$ be an environment. We denote by $||\varphi||^K_\top e$ the set of states in K that satisfy φ, and by $||\varphi||^K_\bot e$ the set of states in K that refute φ. The sets $||\varphi||_\top e$ and $||\varphi||_\bot e$ are defined as follows:*

$$
\begin{array}{lcllcl}
||\mathit{true}||_\top e & = & S & ||\varphi_1 \wedge \varphi_2||_\top e & = & ||\varphi_1||_\top e \cap ||\varphi_2||_\top e \\
||\mathit{true}||_\bot e & = & \emptyset & ||\varphi_1 \wedge \varphi_2||_\bot e & = & ||\varphi_1||_\bot e \cup ||\varphi_2||_\bot e \\
||p||_\top e & = & \{s \mid L(s,p) = \mathsf{t}\} & ||EX\varphi||_\top e & = & ex(||\varphi||_\top e) \\
||p||_\bot e & = & \{s \mid L(s,p) = \mathsf{f}\} & ||EX\varphi||_\bot e & = & ax(||\varphi||_\bot e) \\
||Z||_\top e & = & e(Z) & ||\mu Z \cdot \varphi||_\top e & = & \bigcap\{S' \subseteq S \mid ||\varphi||_\top e[Z \to S'] \subseteq S'\} \\
||Z||_\bot e & = & \overline{e(Z)} & ||\mu Z \cdot \varphi||_\bot e & = & \bigcup\{S' \subseteq S \mid S' \subseteq ||\varphi||_\bot e[Z \to S']\} \\
||\neg\varphi||_\top e & = & ||\varphi||_\bot e & ||\neg\varphi||_\bot e & = & ||\varphi||_\top e
\end{array}
$$

where $ex(S') = \{s \mid \exists s' \in S \cdot R^{must}(s,s') \wedge s' \in S'\}$ and $ax(S') = \{s \mid \forall s' \in S \cdot R^{may}(s,s') \Rightarrow s' \in S'\}$.

For a closed L_μ formula φ, $||\varphi||^K_\lambda e_1 = ||\varphi||^K_\lambda e_2$ for any e_1 and e_2 and $\lambda \in \{\top, \bot\}$. Thus, e can be safely dropped when φ is closed. We also omit K when it is clear from the context. Note that Kripke structures are special cases of KMTSs, i.e., a Kripke structure K is a KMTS where $R^{must} = R^{may}$. Thus, the above semantics applies to Kripke structures, as well.

In this thesis, we often express temporal formulas in CTL which is a fragment of L_μ. The CTL syntax is defined w.r.t. a set AP of atomic propositions as follows:

$$
\begin{array}{rcl}
\varphi & ::= & p \mid \neg p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid EX\varphi \mid AX\varphi \mid E[\varphi U \varphi] \mid \\
& & A[\varphi U \varphi] \mid E[\varphi \tilde{U} \varphi] \mid A[\varphi \tilde{U} \varphi]
\end{array}
$$

where $p \in AP$. The operators AU and EU are universal and existential until operators, respectively; and operators $E\tilde{U}$ and $A\tilde{U}$ are their duals, respectively. Other CTL operators can be defined from these:

$$\begin{array}{rclcrcl} AG\varphi & = & A[\mathsf{false}\ \tilde{U}\varphi] & & EG\varphi & = & E[\mathsf{false}\ \tilde{U}\varphi] \\ AF\varphi & = & A[\mathsf{true}\ U\varphi] & & EF\varphi & = & E[\mathsf{true}\ U\varphi] \end{array}$$

The informal meaning of the CTL temporal operators is: given a state and paths emanating from it, φ holds in one (EX) or all (AX) next states; φ holds in some future state along one (EF) or all (AF) paths; φ holds globally along one (EG) or all (AG) paths, and φ holds until a point where ψ holds along one (EU) or all (AU) paths. Temporal operators EX, EG, and EU together with the propositional connectives form an adequate set (i.e., all other operators can be defined from them).

CTL has a fixpoint characterization which provides a straightforward procedure for translating CTL to L_μ. Thus, the formal semantics of CTL over KMTSs and Kripke structures follows from Definition 2.2.2.

3-valued semantics of KMTS. A KMTS K is *consistent* if $R^{must} \subseteq R^{may}$. For every consistent KMTS K and $\varphi \in L_\mu$, $||\varphi||_\top \cap ||\varphi||_\bot = \emptyset$, i.e., a consistent K does not satisfy $\varphi \wedge \neg\varphi$.

The semantics of L_μ over a consistent KMTS K can be described as a 3-valued function $||.||^K_{\mathbf{3}} : L_\mu \times S \to \mathbf{3}$ as follows,

- $||\varphi||^K_{\mathbf{3}}(s) = \mathsf{t}$ if $s \in ||\varphi||^K_\top$,
- $||\varphi||^K_{\mathbf{3}}(s) = \mathsf{f}$ if $s \in ||\varphi||^K_\bot$, and
- $||\varphi||^K_{\mathbf{3}}(s) = \mathsf{m}$, if $s \notin ||\varphi||^K_\top \wedge \notin ||\varphi||^K_\bot$.

The value of φ in K, denoted $||\varphi||^K_{\mathbf{3}}$, is defined as $||\varphi||^K_{\mathbf{3}}(s_0)$, where s_0 is the initial state of K.

We now present a few example properties over the Kripke structure and the KMTS in Figure 2.1:

Propositional property ($\varphi = p \wedge q$). This property states that both p and q must be true in the initial state. Property φ holds in the KMTS in Figure 2.1(b) because the initial state satisfies both p and q. However, this property does not hold in the model in Figure 2.1(a) because the initial state does not satisfy q.

Modal property ($\varphi = AXq$). This property states that q is true in all the immediate successors of the initial state. Property φ holds in the model in Figure 2.1(a) because all the immediate successors of s_0 satisfy q. This property evaluates to m over the KMTS in Figure 2.1(b) because the only transition that violates this property is a *may*, but not *must* transition. More specifically, φ holds in the *must* fragment of this model, but not in its *may* fragment, making the overall value of φ unknown.

Reachability property ($\varphi = EFq$). This property states that there is a path from the initial state to a state satisfying q. Property φ holds in both models in Figure 2.1 because there is a path from their initial state to a state satisfying q. Note that for the KMTS in Figure 2.1(b), this path is present in both of its *must* and *may* fragments.

Invariance property ($\varphi = AGp$). This property states that p holds in all the states of a model. Property φ does not hold in the model in Figure 2.1(a) because p is false in state s_1, and it evaluates to m in the model in Figure 2.1(b) because the value of p is unknown in state t_1.

2.2.2 Trace Semantics of Action-based Models

In this thesis, we formalize properties over action-based models as *finite positive/negative traces*. These traces can describe *safety* and *finitary liveness* properties. Note that these

traces cannot capture arbitrary liveness, fairness, or mixed safety and liveness properties, and hence, are less expressive than both L_μ and CTL.

Positive traces characterize the behaviours that a model must provide, and negative traces the behaviours that are forbidden (Grosu & Smolka, 2005). For example, consider a property φ_1 which states that action b should follow action a. This property is a positive trace. To satisfy φ_1, the trace $a \cdot b$ should be exhibited by the model. In contrast, consider property φ_2 which states that action c should not follow action a. This property is a negative trace. To satisfy this property, the trace $a \cdot c$ must be excluded from the model traces.

Note that, in principle, negative traces suffice for capturing safety and finitary liveness properties. However, negative traces may sometimes hold vacuously in a model. For example, a model that has an empty or a highly restricted set of behaviours due to deadlock situations may satisfy negative traces vacuously. In such cases, positive traces are helpful because they can describe the require behaviours of a model.

Let $M = (S, s_0, E, R)$ be an LTS, and let $\sigma = e_1 e_2 \dots e_n$ be a trace over E' where $E' \subseteq E$. We say M satisfies a positive trace σ if $\sigma \in \mathcal{L}(M@E')$, and dually, M satisfies a negative trace σ if $\sigma \notin \mathcal{L}(M@E')$. Thus, model checking an LTS against positive/negative traces amounts to solving the language membership problem for finite automata. Here, we formulate this problem in terms of the parallel composition operator. This is similar to the approach taken in the LTSA tool for model checking LTSs (Magee & Kramer, 2006). We first introduce the notion of parallel composition over LTSs, and then describe model checking of LTSs and MixTSs using positive/negative traces.

Composition. The composition of two LTSs that run asynchronously and communicate through synchronous message passing is formalized as *parallel composition* (Milner, 1989). The parallel composition operator $||$ is a commutative and associative operator that combines the behaviours of two LTSs by synchronizing the actions that are present in

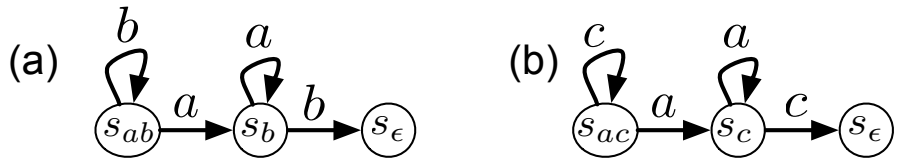


Figure 2.4: Examples of property LTSs: (a) A property LTS for the trace $a \cdot b$, and (b) a property LTS for the trace $a \cdot c$.

both LTSs and interleaving the remaining actions.

Definition 2.2.3 (Parallel Composition) *(Milner, 1989) Let* $M_1 = (S_1, s_0, R_1, E_1)$ *and* $M_2 = (S_2, t_0, R_2, E_2)$ *be LTSs. The* parallel composition *of* M_1 *and* M_2*, denoted* $M_1||M_2$*, is defined as an LTS* $(S_1 \times S_2, (s_0, t_0), R, E_1 \cup E_2)$*, where* R *is the smallest relation satisfying the following:*

$$\begin{aligned} R \;=\; & \{((s,t),e,(s',t)) \mid (s,e,s') \in R_1 \wedge e \notin E_2\} \bigcup \\ & \{((s,t),e,(s,t')) \mid (t,e,t') \in R_2 \wedge e \notin E_1\} \bigcup \\ & \{((s,t),e,(s',t')) \mid (s,e,s') \in R_1 \wedge (t,e,t') \in R_2\} \end{aligned}$$

Model Checking of LTSs. To determine if a positive or negative trace holds over an action-based model, we first translate that trace to an LTS. Formally, let $\sigma = e_0 e_1 \ldots e_n$ be a trace over E'. A *property LTS* M_σ is a tuple (S, s_σ, R, E') where

$$\begin{aligned} S = & \{s_{\sigma'} \mid \sigma' \text{ is a (possibly empty) suffix of } \sigma\} \\ R = & \{(s_{\sigma'}, e, s_{\sigma''}) \mid \sigma' = e.\sigma'' \wedge \ \sigma' \text{ is a suffix of } \sigma\} \bigcup \\ & \{(s_{\sigma'}, e', s_{\sigma'}) \mid \sigma' = e.\sigma'' \wedge e' \in E' \wedge e \neq e' \wedge \ \sigma' \text{ is a suffix of } \sigma\} \end{aligned}$$

For example, Figure 2.4(a) shows the property LTS for the trace $\sigma = a \cdot b$ and Figure 2.4(b) – the property LTS for the trace $\sigma = a \cdot c$. Note that state s_ϵ, which corresponds to the empty suffix ϵ, is without outgoing transitions in every property LTS. Reachability of this state determines whether M can generate σ. That is, an LTS M generates σ iff state s_ϵ is reachable in $M_\sigma||M$, and M does not generate σ iff state s_ϵ

is not reachable in $M_\sigma || M$. Thus, model checking an LTS M against a positive or a negative trace σ can be done by composing M with M_σ and checking the reachability of s_ϵ. For example, it can be seen that the LTS in Figure 2.3(a) satisfies the positive trace $a \cdot b$ because this LTS is can generate this positive trace. This LTS violates the negative trace $a \cdot c$ because its language includes this negative trace.

Model checking of MixTSs. To describe the semantics of positive and negative traces over MixTSs, we note that MixTSs are not classical transition systems, and hence, their semantics is not boolean. Similar to KMTSs, a MixTS is *consistent* when $R^{must} \subseteq R^{may}$. The semantics of positive/negative traces over consistent MixTSs is 3-valued. The value of a positive trace σ over a MixTS M is as follows:

- σ is t over M if M^{must} satisfies σ
- σ is f over M if M^{may} does not satisfy σ
- σ is m over M if M^{must} does not satisfy σ, but M^{may} satisfies σ.

Dually, the value of a negative trace σ over a MixTS M is as follows:

- σ is t over M if M^{may} satisfies σ
- σ is f over M if M^{must} does not satisfy σ
- σ is m over M if M^{must} satisfies σ, but M^{may} does not satisfies σ.

For example, the consistent MixTS in Figure 2.3(b) satisfies the positive trace $a \cdot b$ because its *must* fragment can generate this trace. However, the value of the negative trace $a \cdot c$ over this model is m because its *may* fragment can generate this negative behaviour. Note that for consistent MixTSs, we have $\mathcal{L}(M^{must}) \subseteq \mathcal{L}(M^{may})$. Thus, it never happens that a trace is generated by M^{must}, but not by M^{may}. The above semantics shows that model checking of positive/negative traces over MixTSs can be done via two calls to a classical

LTS model-checker: One for checking M^{must}, and the other for checking M^{may}. This allows us to implement a model checker for MixTSs using existing LTS model checkers, e.g., the LTSA tool (Magee & Kramer, 2006).

2.3 Behaviour Preserving Relations

In this section, we describe behaviour preserving relations over the formalisms introduced in Section 2.1. We then present some theorems, showing how these relations can preserve semantic properties of behavioural models.

2.3.1 Relations over State-based Formalisms

From the point of view of a temporal logic, two models are equivalent if there does not exist any formula that can distinguish between them. The relation that structurally characterizes this equivalence is known as *bisimulation* relation.

Definition 2.3.1 (Bisimulation) *(Milner, 1989) Let K_1 and K_2 be Kripke structures. A* bisimulation relation $\rho \subseteq S_1 \times S_2$ *is the largest relation where $\rho(s,t)$ iff*

1. $L(s) = L(t)$
2. $\forall s' \in S_1 \cdot R_1(s,s') \Rightarrow \exists t' \in S_2 \cdot R_2(t,t') \wedge \rho(s',t')$
3. $\forall t' \in S_2 \cdot R_2(t,t') \Rightarrow \exists s' \in S_1 \cdot R_2(s,s') \wedge \rho(s',t')$

We say K_1 is bisimilar to K_2 and write $K_1 \equiv K_2$ if there is a bisimulation relation ρ such that $\rho(s_0,t_0)$ where s_0 and t_0 are the initial states of K_1 and K_2, respectively.

It is known that L_μ is a logical characterization of bisimulation (Milner, 1989). In other words, states s and t are bisimilar if and only if they satisfy the same L_μ properties.

Since bisimulation is an equivalence relation, it cannot relate models defined at different levels of abstraction or models that capture different amounts of knowledge. Such

relations should be formalized by *preorders* ($\preceq$) rather than by equivalences ($\equiv$). *Refinement relation* defined over KMTSs is an example of a preorder that captures the notion of "more defined than" relation between a pair of KMTSs (Larsen & Thomsen, 1988).

Definition 2.3.2 (Refinement) *(Larsen & Thomsen, 1988) Let K_1 and K_2 be KMTSs where $AP_1 = AP_2 = AP$. A* refinement relation $\rho \subseteq S_1 \times S_2$ *is the largest relation where $\rho(s,t)$ iff*

1. $\forall p \in AP \cdot L_1(s,p) \preceq L_2(t,p)$
2. $\forall s' \in S_1 \cdot R_1^{must}(s,s') \Rightarrow \exists t' \in S_2 \cdot R_2^{must}(t,t') \wedge \rho(s',t')$
3. $\forall t' \in S_2 \cdot R_2^{may}(t,t') \Rightarrow \exists s' \in S_1 \cdot R_1^{may}(s,s') \wedge \rho(s',t')$

We say K_2 refines K_1 and write $K_1 \preceq K_2$, if there is a refinement ρ such that $\rho(s_0,t_0)$, where s_0 and t_0 are the initial states of K_1 and K_2, respectively.

Intuitively, t refines s if the variables in s are less defined than those in t (condition 1); every *must* transition from s is matched by some *must* transition from t (condition 2); and every *may* transition from t is matched by some *may* transition from s (condition 3).

Refinement preserves L_μ formulas (Huth *et al.*, 2001). This is because if K_2 refines K_1 then the *must* behaviors of K_1 are a subset of the *must* behaviors of K_2; and the *may* behaviors of K_2 are a subset of the *may* behaviors of K_1.

Theorem 2.3.1 *(Huth* et al.*, 2001) Let K_1 and K_2 be KMTSs. If $K_1 \preceq K_2$, then*

$$\forall \varphi \in L_\mu \cdot ||\varphi||_{\mathbf{3}}^{K_1} \preceq ||\varphi||_{\mathbf{3}}^{K_2}$$

The above theorem implies that if any L_μ property is t (resp. f) over K_1, then it is also t (resp f) over K_2. However, if a property evaluates to m over K_1, it may or may not keep its value over K_2. Note that since CTL is a fragment of L_μ, the above preservation result holds for CTL as well.

2.3.2 Relations over Action-based Formalisms

Refinement between two LTSs at different levels of abstraction is usually formalized by *simulation* or a variant of simulation. In this thesis, we use the notion of *weak simulation* (also known as observational simulation) to check the existence of a refinement relation between two LTSs (Milner, 1989). This notion can be used for relating LTSs with different sets of actions by replacing their non-shared actions with τ. For states s and s' of an LTS M, we write $s \stackrel{\tau}{\Longrightarrow} s'$ to denote $s(\stackrel{\tau}{\longrightarrow})^* s'$. For $e \neq \tau$, we write $s \stackrel{e}{\Longrightarrow} s'$ to denote $s(\stackrel{\tau}{\Longrightarrow})(\stackrel{e}{\longrightarrow})(\stackrel{\tau}{\Longrightarrow})s'$.

Definition 2.3.3 (Simulation) *(Milner, 1989) Let M_1 and M_2 be LTSs, where $E_1 = E_2 = E$. A relation $\rho \subseteq S_1 \times S_2$ is a* weak simulation*, or* simulation *for short, where $\rho(s,t)$ iff*

1. $\forall s' \in S_1 \cdot \forall e \in E \cup \{\tau\} \cdot s \stackrel{e}{\longrightarrow} s' \Rightarrow \exists t' \in S_2 \cdot t \stackrel{e}{\Longrightarrow} t' \wedge \rho(s', t')$

We say M_2 simulates M_1, written $M_1 \preceq M_2$, if there is a simulation ρ such that $\rho(s_0, t_0)$, where s_0 and t_0 are the initial states of M_1 and M_2, respectively.

Theorem 2.3.2 *(Milner, 1989) Let M_1 and M_2 be LTSs where $M_1 \preceq M_2$. Then, $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$.*

Based on the above theorem, simulation is a sufficient condition for trace containment. This shows that if M_1 satisfies a *positive* trace, so does M_2; and dually, if M_2 satisfies a *negative* trace, so does M_1. Recall that $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$ capture only the *observable* behaviours of M_1 and M_2. Thus, Theorem 2.3.2 states that if $M_1 \preceq M_2$, then M_2 can generate every observable trace of M_1, but not necessarily traces with τ-steps.

The notion of refinement over MixTSs is defined using two dual simulation relations: One relating the *must* fragments, and the other – the *may* fragments. For states s and s' of a MixTS M, we write $s \stackrel{e}{\Longrightarrow}^{must} s'$ to denote $s \stackrel{e}{\Longrightarrow} s'$ in M^{must}, and $s \stackrel{e}{\Longrightarrow}^{may} s'$ to denote $s \stackrel{e}{\Longrightarrow} s'$ in M^{may}.

Definition 2.3.4 (Refinement) *(Larsen & Thomsen, 1988) Let M_1 and M_2 be MixTSs. A relation $\rho \subseteq S_1 \times S_2$ is a* refinement*, where $\rho(s,t)$ iff*

1. $\forall s' \in S_1 \cdot \forall e \in E_1 \cup \{\tau\} \cdot s \xrightarrow{e}^{must} s' \Rightarrow \exists t' \in S_2 \cdot t \stackrel{e}{\Longrightarrow}^{must} t' \wedge \rho(s',t')$

2. $\forall t' \in S_2 \cdot \forall e \in E_2 \cup \{\tau\} \cdot t \xrightarrow{e}^{may} t' \Rightarrow \exists s' \in S_1 \cdot s \stackrel{e}{\Longrightarrow}^{may} s' \wedge \rho(s',t')$

We say M_2 refines M_1, written $M_1 \preceq M_2$, if there is a refinement ρ such that $\rho(s_0,t_0)$, where s_0 and t_0 are the initial states of M_1 and M_2, respectively.

Theorem 2.3.3 *(Larsen & Thomsen, 1988) Let M_1 and M_2 be MixTSs where $M_1 \preceq M_2$. Then, $\mathcal{L}(M_1^{must}) \subseteq \mathcal{L}(M_2^{must})$, and $L(M_2^{may}) \subseteq \mathcal{L}(M_1^{may})$.*

The above theorem implies that if any positive/negative trace is t (resp. f) over M_1, then it is also t (resp f) over M_2. However, if a trace evaluates to m over M_1, it may or may not keep its value over M_2.

2.4 Summary

In this chapter, we presented both state-based and action-based modelling formalisms, as well as some background information on the semantics of these formalisms and the behaviour-preserving relations defined over these formalisms.

We use the concepts introduced in this chapter throughout this thesis. Specifically, in Chapter 3, we concentrate on state-based models, and use KMTSs to formalize partial descriptions of individual features. In Chapters 4, we use parameterized state machines, inspired by MixTSs, to describe variant feature specifications. Finally, in Chapter 5, we use I/O automata, an extension of LTSs, to analyse interactions between models describing different system features.

Chapter 3

Merging Partial Behavioural Models

3.1 Introduction

In this chapter, we focus on the problem of merging *partial* and *consistent* models of an individual feature with the goal of producing a more comprehensive description of that feature. Our merging approach is defined for state machine models describing the same feature of a system from different perspectives and possibly using different vocabulary. To unify the vocabulary of these state machines, we formalize them as partial state machines with 3-valued semantics. We propose a merge procedure that automatically determines whether a pair of partial state machines are consistent, and if so, computes their merge. We illustrate our merging approach using two complementary perspectives on a photo-taking functionality of a camera, and report on an implementation of our approach.

3.1.1 Motivating Example

We illustrate the problem of merging partial models on a pair of specification models of the photo-taking feature of a camera[1]. To take a photo, a user needs to press the *shutter*

[1]The example is adapted from (Sabetzadeh & Easterbrook, 2003).

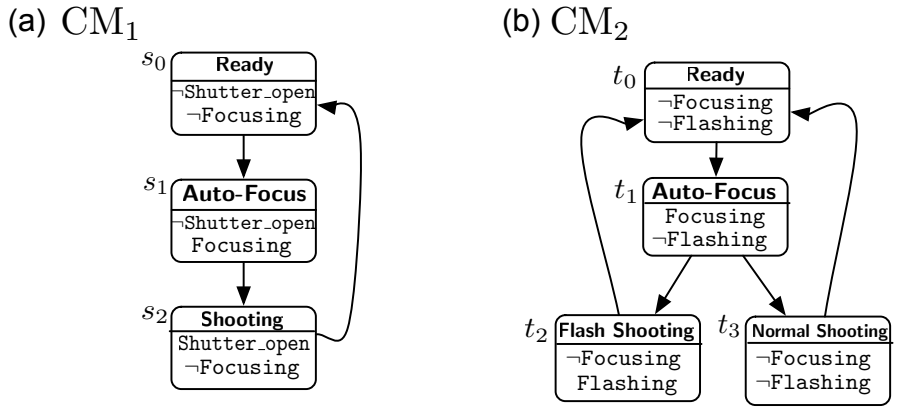


Figure 3.1: Models of the photo-taking feature of a camera: (a) $\mathbf{CM_1}$ and (b) $\mathbf{CM_2}$.

button half-way. When *focus* is achieved, the shutter button can be pressed completely to take the picture. Under low-light conditions, the built-in *flash* should fire automatically.

Two different specification models of a camera, $\mathbf{CM_1}$ and $\mathbf{CM_2}$, are shown in Figure 3.1. The goal of $\mathbf{CM_1}$ is to specify the focusing feature and the behaviour of the camera's shutter. In the first state of this model (s_0), the shutter is closed and the focus is not yet achieved; in the second (s_1), the focus is achieved; and in the third (s_2), the shutter becomes open so that the photo can be taken. Like $\mathbf{CM_1}$, model $\mathbf{CM_2}$ (see Figure 3.1(b)) describes the focusing feature. In addition to focusing, $\mathbf{CM_2}$ also describes the built-in flash of the camera. The flash is disabled in the first and second states (t_0 and t_1). After state t_1, depending on the light intensity, the flash is either fired (state t_2) or remains disabled (state t_3).

In this example, we assume vocabulary consistency, i.e., there is no name clash, and no two distinctly named propositions represent the same thing. This means that these two models use `Focusing`, `Shutter_open`, and `Flashing` to represent whether the focus is achieved, whether the camera's shutter is open, and whether the flash is enabled, respectively. We refer to the set of propositions used by a model as its *vocabulary* (e.g., $\{\texttt{Focusing}, \texttt{Shutter\_open}\}$ for $\mathbf{CM_1}$), and the union of all vocabularies as the *unified set of propositions*. In general, achieving and maintaining vocabulary consistency is a

Property	Description	CTL formulation
$\mathbf{P_1}$	Whenever focus is achieved, we can take a picture.	$AG(\texttt{fo} \Rightarrow EX\texttt{s})$
$\mathbf{P_2}$	We cannot take a picture without achieving focus.	$\neg E[\neg\texttt{fo}\ U\ \texttt{s}]$
$\mathbf{P_3}$	After focus is achieved, we take a photo (with or without flash).	$AG\big(\texttt{fo} \Rightarrow (EX\texttt{fl} \wedge EX\neg\texttt{fl} \wedge AX\texttt{s})\big)$
$\mathbf{P_4}$	Camera's flash cannot fire while camera is trying to achieve focus.	$AG(\neg\texttt{fo} \vee \neg\texttt{fl})$
$\mathbf{P_5}$	Whenever flash is enabled, shutter is open.	$AG(\texttt{fl} \rightarrow \texttt{s})$

Table 3.1: Properties of the camera models.

difficult problem, studied, e.g., by (Gruber, 1991). We consider this issue to be orthogonal to the techniques presented in this thesis.

In the rest of this chapter, we abbreviate the propositions Focusing, Flashing, and Shutter_open by fo, fl and s, respectively. We denote the set $\{\texttt{fo}, \texttt{fl}, \texttt{s}\}$, the unified set of propositions for the camera model, by AP_u.

State-machine models are typically constructed to ensure that the resulting design satisfies (or violates) certain properties. For example, some properties of the camera example are shown in Table 3.1. These properties are either representations of individual executions of the system, such as *use cases* or *scenarios* (e.g., $\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_3}$), or statements about *all* system executions, such as invariants (e.g., $\mathbf{P_4}$ and $\mathbf{P_5}$). $\mathbf{P_1}$ and $\mathbf{P_3}$ are positive scenarios whereas $\mathbf{P_2}$ is a negative scenario: it prohibits behaviours where state **Shooting** is immediately followed by state **Ready** in Figure 3.1. It is easy to see that $\mathbf{CM_1}$ satisfies $\mathbf{P_1}$ and $\mathbf{P_2}$, and $\mathbf{CM_2}$ satisfies $\mathbf{P_4}$. However, we cannot determine the value of $\mathbf{P_3}$ and $\mathbf{P_5}$ because it uses two propositions fl and s, each of which is present in only one of $\mathbf{CM_1}$ and $\mathbf{CM_2}$.

3.1.2 Contributions of This Chapter

Models $\mathbf{CM_1}$ and $\mathbf{CM_2}$ provide two complementary descriptions of the camera's photo-taking functionality: They both describe the focusing feature (using proposition `fo`), but $\mathbf{CM_1}$ additionally describes the shutter button (using proposition `s`), and $\mathbf{CM_2}$ – the built-in flash (using proposition `fl`). To be able to evaluate properties involving non-shared propositions, like $\mathbf{P_3}$ and $\mathbf{P_5}$, we need to merge $\mathbf{CM_1}$ and $\mathbf{CM_2}$. Computing such a merged model involves answering several questions. Particularly,

- How can we unify the vocabulary of different models?
- How can we evaluate properties containing non-shared propositions over each individual model?
- How can we determine if the models are consistent?
- How can we merge consistent models?

In this chapter, we answer these questions for state machine models. The key argument in our work is that the classical semantics of state machines is not suitable for describing complementary perspectives of system features. Under this semantics, it is not clear how queries containing non-shared propositions should be evaluated. To address this problem, we use state machines with partial semantics (or partial state machines). In partial state machines, propositions can accept the value *unknown* (or *maybe*) in addition to true and false. This allows us to extend the vocabulary of each individual model by adding non-shared propositions as *maybe* propositions to every state of that model. After extending the vocabulary of a model, we can verify it against every property defined over AP_u; however, some properties may evaluate to *maybe* because the model may not be elaborate enough to generate a conclusive result for these properties. The *maybe* properties *often* indicate the behaviours that are left unspecified in one model and can be refined by merging that model with other perspectives. Note that sometimes properties

evaluate to *maybe* over partial models not because the models do not contain sufficient information for conclusive evaluation of properties of interest, but because of the inherent imprecision of the logic that we use to specify the incompleteness, i.e., 3-valued logic (Godefroid & Jagadeesan, 2002; Godefroid & Huth, 2005; Gurfinkel & Chechik, 2005; Nejati *et al.*, 2006).

3.1.3 Organization of This Chapter

The rest of this chapter is organized as follows: In Section 3.2, we propose a merge procedure that automatically determines whether two input state machines are consistent, and if so, computes their merge. In Section 3.3, we discuss the implementation of our procedure. We compare the approach proposed in this chapter with related work in Section 3.4, and summarize the chapter in Section 3.5.

3.2 Merging Partial Consistent Models

In this section, we look at the problem of merging partial and consistent state machines. Section 3.2.1 provides background on partial state machine formalisms and introduces a logical notion of consistency between them. Section 3.2.2 describes our approach to merging consistent and partial models. Similar to the prior work on merging behavioural models (Uchitel & Chechik, 2004), we define merge as a common refinement of the input models; however, in this chapter, we consider state-based models, whereas in (Uchitel & Chechik, 2004), action-based formalisms were studied.

3.2.1 Partial Models and Consistency

Requirements models are inherently incomplete. Each model can only focus on a few features of a system, and thus, uses just a fraction of the unified set of propositions. For example, $\mathbf{CM_1}$ (see Figure 3.1(a)) does not address the built-in flash feature, and hence,

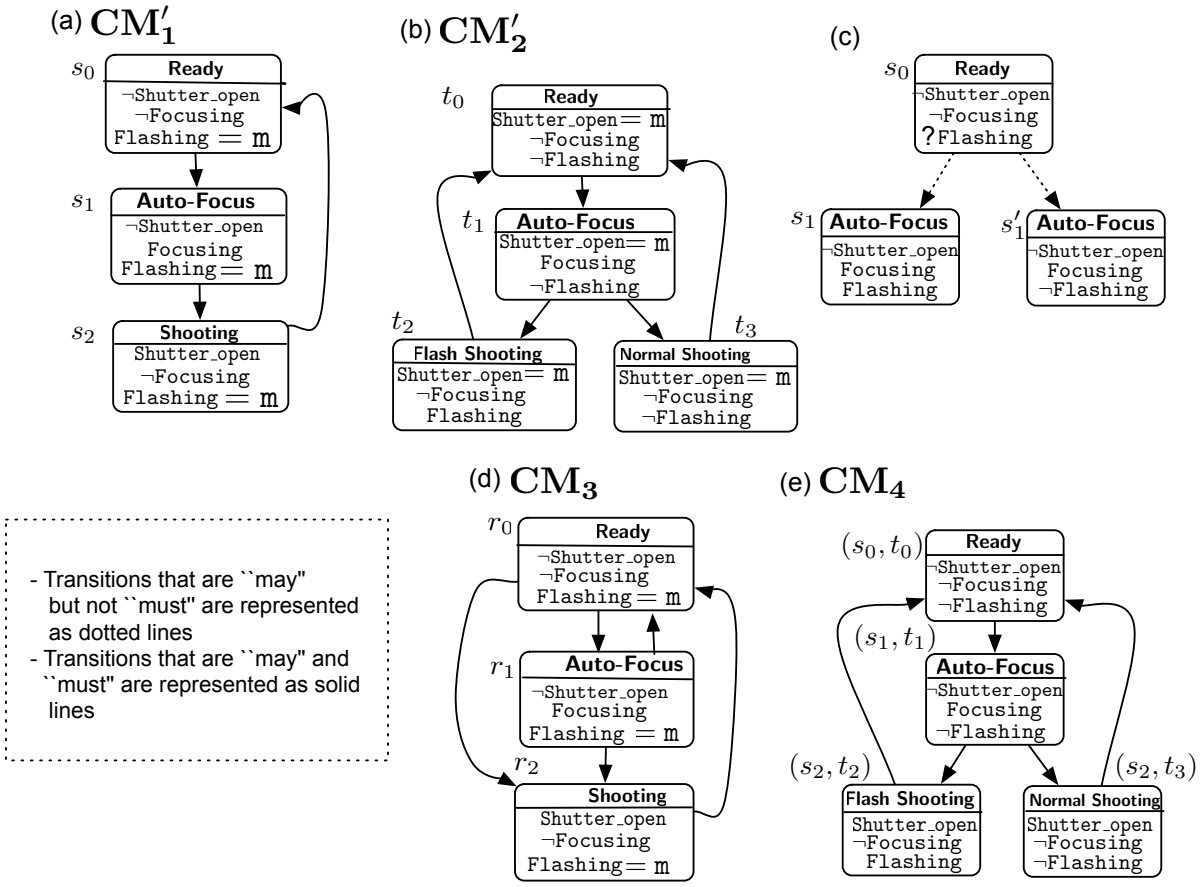


Figure 3.2: Example models: (a) $\mathbf{CM_1}'$ which is $\mathbf{CM_1}$ with the vocabulary $\{\mathtt{s}, \mathtt{fo}, \mathtt{fl}\}$; (b) $\mathbf{CM_2}'$ which is $\mathbf{CM_2}$ with the vocabulary $\{\mathtt{s}, \mathtt{fo}, \mathtt{fl}\}$; (c) a model with partial transitions, i.e., transitions in $R^{may} \setminus R^{must}$; (d) a camera description inconsistent with $\mathbf{CM_1}$ and $\mathbf{CM_2}$; and (e) $\mathbf{CM_4}$, the merge of $\mathbf{CM_1}$ and $\mathbf{CM_2}$.

does not use the proposition `fl`.

To be able to compare and merge incomplete models, we need to unify the sets of vocabulary of these models and address the resulting incompleteness. We do so using 3-valued logic (Kleene, 1952) (See Figure 2.2). This logic has been used by several researchers to model and reason with incompleteness and uncertainty (e.g., (Hussain & Huth, 2004; Huth & Pradhan, 2001; Uchitel & Chechik, 2004; Bruns & Godefroid, 2000; Chechik *et al.*, 2003)). It extends classical logic with an additional truth value, denoted by maybe (m). For example, when the set of vocabulary of $\mathbf{CM_1}$ is extended to AP_u, we

simply set the missing proposition `fl` to `m` in all of the states of $\mathbf{CM_1}$. The result is shown in Figure 3.2(a). Similarly, the set of vocabulary of $\mathbf{CM_2}$ is extended to AP_u by setting the missing proposition `s` to `m` in all the states of $\mathbf{CM_2}$ (see Figure 3.2(b)).

We formalize our models as *Kripke Modal Transition Systems (KMTSs)* (see Definition 2.1.2). For every KMTS $K = (S, s_0, R^{must}, R^{may}, L, AP)$, we assume that the set AP of atomic propositions is a subset of the unified set of propositions, i.e., $AP \subseteq AP_u$. To compare models with different variable sets, we assume that the labelling function L for every state machine is defined for every variable in AP_u, and further, for every $p \in AP_u \setminus AP$ and every $s \in S$, $L(s, p) \triangleq \mathsf{m}$.

Models $\mathbf{CM_1}'$ and $\mathbf{CM_2}'$ in Figure 3.2(a) and (b) do not have partial transitions, i.e., for these models, we have $R^{must} = R^{may}$. An example of a model with partial transitions, i.e., transitions in $R^{may} \setminus R^{must}$, is shown in Figure 3.2(c). This model is a fragment of a camera model where transitions from s_0 to s_1 and s_1' are partial, representing the fact that this model is not sure about the status of the camera's flash (`fl`) after the initial state s_0. From the theoretical point of view, adding partial transitions to models with maybe propositions does not add any expressive power: In (Godefroid & Jagadeesan, 2003), it is shown that three well-known formalisms for partial models, i.e., Partial Kripke Structures, Modal Transition Systems and Kripke Model Transition Systems, are all equally expressive. Specifically, they showed that KMTSs with maybe transitions can be translated to KMTSs without any maybe transitions such that the translation preserves the refinement ordering of the KTMSs. Even though partial transitions do not change the expressiveness, they can result in models with fewer states, and hence, we may want to use them for creating more concise models. In our examples, we use the following convention for transitions: transitions represented by solid arrows are in $R^{must} \cap R^{may}$, and transition represented by dotted arrows are in $R^{may} \setminus R^{must}$.

We use refinement relations $\preceq$ given in Definition 2.3.2 to relate KMTSs. For example, model $\mathbf{CM_4} = (S_\mathbf{c}, ...)$ in Figure 3.2(e) is a refinement of model $\mathbf{CM_1}'$ in Figure 3.2(a),

where the refinement relation is $\{(s,(s,x)) \mid (s,x) \in S_{\mathbf{c}}\}$. Note that our treatment of the labelling function allows us to relate models with different variable sets.

Refinement preserves all definite behaviours of the original model. Furthermore, it preserves truth and falsity of properties expressed in the temporal logic L_μ (μ-calculus) such as the properties described in Table 3.1 (Huth *et al.*, 2001) (see Theorem 2.3.1). Note that refinement preserves valuation of not only positive (e.g., $\mathbf{P_1}$, $\mathbf{P_3}$, $\mathbf{P_4}$, and $\mathbf{P_5}$) but also negative (e.g., $\mathbf{P_2}$), universal (e.g. $\mathbf{P_4}$ and $\mathbf{P_5}$), existential, and mixed (e.g. $\mathbf{P_1}$ and $\mathbf{P_3}$) properties.

We now aim to characterize similarities between models which are not necessarily refinements of each other.

Definition 3.2.1 (Common Refinement) *(Uchitel & Chechik, 2004) Let K_1 and K_2 be KMTSs. A KMTS K_3 is a* common refinement *of K_1 and K_2 iff $K_1 \preceq K_3$ and $K_2 \preceq K_3$. Furthermore, K_3 is the* least common refinement *iff for every common refinement K_4, $K_3 \preceq K_4$.*

Like refinement, common refinement preserves truth and falsity of properties expressed in L_μ (Huth *et al.*, 2001).

Theorem 3.2.1 *Let K_3 be a common refinement of K_1 and K_2. Then, $\forall \varphi \in L_\mu$:*

$$(||\varphi||^{K_1} = \mathsf{t}) \ \vee \ (||\varphi||^{K_2} = \mathsf{t}) \ \Rightarrow \ ||\varphi||^{K_3} = \mathsf{t}$$
$$(||\varphi||^{K_1} = \mathsf{f}) \ \vee \ (||\varphi||^{K_2} = \mathsf{f}) \ \Rightarrow \ ||\varphi||^{K_3} = \mathsf{f}$$

Moreover, if K_3 is the least common refinement of K_1 and K_2, then for every common refinement K_4,

$$||\varphi||^{K_4} = \mathsf{m} \Rightarrow ||\varphi||^{K_3} = \mathsf{m}$$

Proof:

The proof follows from Theorem 2.3.1: if either K_1 or K_2 satisfies any $\varphi \in L_\mu$, so does K_3, and if neither K_1 or K_2 satisfies any $\varphi \in L_\mu$, nor does K_3. □

Note that the other direction of Theorem 3.2.1 does not necessarily hold. That is, $||\varphi||^{K_3} = \mathsf{t}$ may not imply $||\varphi||^{K_1} = \mathsf{t}$ or $||\varphi||^{K_2} = \mathsf{t}$. Similarly, $||\varphi||^{K_3} = \mathsf{f}$ may not imply $||\varphi||^{K_1} = \mathsf{f}$ or $||\varphi||^{K_2} = \mathsf{f}$. For example, $\mathbf{CM_4}$ in Figure 3.2(e) is a common refinement of $\mathbf{CM_1}'$ and $\mathbf{CM_2}'$ in Figures 3.2(a) and (b), respectively. Property $\mathbf{P_5}$ in Table 3.1 holds on $\mathbf{CM_4}$ but it neither holds on $\mathbf{CM_1}'$ nor on $\mathbf{CM_2}'$.

By Theorem 3.2.1, every common refinement preserves all definite properties of the input models, i.e., properties that evaluate to true or false in at least one of the input models are preserved in every common refinement. Therefore, every common refinement is *sound*. However, common refinements, even the least one, are not precise. In particular, the least common refinement may not preserve the maybe properties of the input models, i.e., a property that evaluates to maybe in both models is not necessarily maybe in the least common refinement. For this reason, in Theorem 3.2.1, we cannot say that for the least common refinement K_4, the following holds

$$||\varphi||^{K_4} = \mathsf{m} \Rightarrow ||\varphi||^{K_1} = \mathsf{m} \wedge ||\varphi||^{K_2} = \mathsf{m}$$

For example, in Figure 3.2, it can be shown that $\mathbf{CM_4}$ is the least common refinement of $\mathbf{CM_1}'$ and $\mathbf{CM_2}'$, but the property $\mathbf{P_5}$ evaluates to maybe on both $\mathbf{CM_1}'$ and $\mathbf{CM_2}'$, and to true on $\mathbf{CM_4}$. As discussed in Section 3.1.2, this problem is due to the inherent imprecision of the 3-valued semantics. To resolve this problem, we need to change the 3-valued semantics and use thorough semantics instead (Godefroid & Jagadeesan, 2002; Godefroid & Huth, 2005; Gurfinkel & Chechik, 2005; Nejati *et al.*, 2006). But, the model checking procedure for thorough semantics is very expensive, and hence, this semantics is impractical in general (Godefroid & Huth, 2005). In short, under the conventional 3-valued semantics of partial models, we cannot say that the least common refinement can preserve all properties of the input models, because it does not necessarily preserve the maybe properties. We can only say that the least common refinement is more precise than other common refinements.

If for some property $\varphi \in L_\mu$, $||\varphi||^{K_1}$ is t and $||\varphi||^{K_2}$ is f, then the common refinement of K_1 and K_2 cannot be described as a KMTS; in this case K_1 and K_2 are called *inconsistent*.

Definition 3.2.2 (Consistency) *KMTSs K_1 and K_2 are* consistent *if*

$$\forall \varphi \in L_\mu \cdot (||\varphi||^{K_1} = \mathsf{t} \Rightarrow ||\varphi||^{K_2} \neq \mathsf{f}) \wedge (||\varphi||^{K_2} = \mathsf{t} \Rightarrow ||\varphi||^{K_1} \neq \mathsf{f})$$

Otherwise, K_1 and K_2 are inconsistent.

Note that consistent partial models K_1 and K_2 may not always produce the same truth values, i.e., a property may evaluate to t or f on one model and m on the other. For example, consider consistent models $\mathbf{CM_1}'$ and $\mathbf{CM_2}'$ in Figure 3.2. $\mathbf{P_4}$ evaluates to m on $\mathbf{CM_1}'$ and to t on $\mathbf{CM_2}'$. In the next section, we define and formalize the notion of merge for consistent models.

3.2.2 Computing Merge

The goal of merging consistent models is to combine (partial) knowledge coming from individual models while preserving all of their agreements. The notion of common refinement underlies this intuition as it captures the "more complete than" relation between two incomplete models (Uchitel & Chechik, 2004). Thus, we define merge using this notion.

Definition 3.2.3 (Merge) *A* merge *of two KMTSs is their common refinement.*

Basing the notion of merge on a common refinement is standard (Uchitel & Chechik, 2004; Huth & Pradhan, 2001; Hussain & Huth, 2004). By Theorem 3.2.1, the least common refinement is the most precise merge. However, the least common refinement of a pair of KMTSs, even if the KMTSs are consistent, is not necessarily a KMTS. In (Schmidt, 2004), it is shown that the family of partial models in which $R^{must} \subseteq R^{may}$, i.e., KMTSs and MTSs (Larsen & Thomsen, 1988), cannot form a complete lattice with

respect to refinement ordering. This is because the least common refinement of a pair of KMTSs can have *must* transitions that are not *may*. Such partial models in which $R^{must} \not\subseteq R^{may}$ are known as *mixed transition systems* (Dams *et al.*, 1997). The reason that the least common refinement of a pair of KMTSs can become a mixed transition system is that to build a least common refinement of KMTSs K_1 and K_2, we can lift any *must* transition in K_1 or K_2 to the merge without any change, but a *may* transition in K_1 (resp. K_2) should be first matched and refined with respect to its corresponding *may* transition in K_2 (resp. K_1), and then lifted to the merge. Thus, in the merged model, we may have *must* transitions that are distinct from *may* transitions because the successors of *must* transitions are less refined than those of the *may* transitions (see (Schmidt, 2004),(Nejati, 2005),(Wei *et al.*, 2008)).

Even though we can always express the least common refinements of KMTSs as mixed transition systems, these least common refinements are often very large and unintuitive. Thus, we would like to focus on formalisms in which $R^{must} \subseteq R^{may}$. Unfortunately, it turns out that for such formalisms the least common refinement may not be unique, i.e., for a pair of KMTSs, we may have several incomparable minimal common refinements described as KMTSs. In (Larsen *et al.*, 1995), it is shown that KMTSs that are *independent* are guaranteed to have least common refinements expressible as KMTS. However, the notion of independence is stronger than consistency, i.e., there are KMTSs that are consistent but not independent. (Fischbein & Uchitel, 2008) improve the work of (Larsen *et al.*, 1995) by providing a merge algorithm that can compute a unique merge for KMTSs that are not necessarily independent. However, since not all KMTSs have a least common refinement (Schmidt, 2004), there are still KMTSs for which the algorithm of (Fischbein & Uchitel, 2008) cannot compute a merge. Thus, in this chapter, we relax the notion of merge, and define it as a common refinement that may not be necessarily the least one.

Clearly, Definition 3.2.3 only applies to consistent models because inconsistent ones do not have a common refinement expressible in 3-valued logics. So, our first goal is to

determine whether two models are consistent. We define consistency recursively, in the same way that other property preserving relations over state machines such as bisimulation and refinement are defined (see Section 2.3.1).

Definition 3.2.4 (Consistency Relation) *Let K_1 and K_2 be KMTSs. We define a* consistency relation $\sim \subseteq S_1 \times S_2$ *where $s \sim t$ iff:*

1. $\forall p \in AP_u \cdot (L_1(s,p) = \mathsf{t} \Rightarrow L_2(t,p) \neq \mathsf{f}) \wedge (L_2(t,p) = \mathsf{t} \Rightarrow L_1(s,p) \neq \mathsf{f})$

2. $\forall s' \in S_1 \cdot R_1^{must}(s,s') \Rightarrow \exists t' \in S_2 \cdot R_2^{may}(t,t') \wedge s' \sim t'$

3. $\forall t' \in S_2 \cdot R_2^{must}(t,t') \Rightarrow \exists s' \in S_1 \cdot R_1^{may}(s,s') \wedge s' \sim t'$

We say K_1 and K_2 are consistent, written as $K_1 \sim K_2$, iff $s_0 \sim t_0$.

Theorem 3.2.2 *K_1 and K_2 have a common refinement iff $K_1 \sim K_2$.*

Proof:

$\Rightarrow$ Let K_3 be a common refinement of K_1 and K_2. Then, there are two refinement relations $\preceq$ and $\preceq'$ s.t. $K_1 \preceq K_3$ and $K_2 \preceq' K_3$. We define a relation $\rho \subseteq S_1 \times S_2$ as follows:

$$\rho = \{(s,t) \in S_1 \times S_2 \mid \exists r \in S_3 \cdot s \preceq r \wedge t \preceq' r\}$$

Informally, ρ contains tuples (s,t) where s and t have a common refinement r. Note that ρ is non-empty because the initial state of K_3, r_0, has to refine both initial states of K_1 and K_2. Thus, $(s_0,t_0) \in \rho$.

We show that ρ is a consistency relation between K_1 and K_2, i.e., ρ satisfies the conditions of Definition 3.2.4:

1. We show that $\forall p \in AP_u \cdot L_1(s,p) = \mathsf{t} \Rightarrow L_2(t,p) \neq \mathsf{f}$ the second part of this condition can be proven in the same way.

$$\begin{aligned} & \forall p \in AP_u \cdot L_1(s,p) = \mathsf{t} \\ \Rightarrow \quad & \text{(Since } s \preceq r\text{)} \\ & L_3(r,p) = \mathsf{t} \\ \Rightarrow \quad & \text{(Since } t \preceq r\text{)} \\ & L_2(t,p) \neq \mathsf{f} \end{aligned}$$

2. To show that $\rho = \sim$, we need to show that $\forall s' \in S_1 \cdot R_1^{must}(s,s') \Rightarrow \exists t' \in S_2 \cdot R_2^{may}(t,t') \wedge (s',t') \in \rho$

$$\begin{aligned} & \forall s' \in S_1 \cdot R_1^{must}(s,s') \\ \Rightarrow \quad & \text{(Since } s \preceq r\text{)} \\ & \exists r' \in S_3 \cdot R_3^{must}(r,r') \wedge s' \preceq r' \\ \Rightarrow \quad & \text{(Since } K_3 \text{ is a partial Kripke structure)} \\ & \exists r' \in S_3 \cdot R_3^{may}(r,r') \wedge s' \preceq r' \\ \Rightarrow \quad & \text{(Since } t \preceq r\text{)} \\ & \exists t' \in S_2 \cdot R_2^{may}(t,t') \wedge t' \preceq r' \\ \Rightarrow \quad & \text{(By definition of } \rho\text{)} \\ & \exists t' \in S_2 \cdot R_2^{may}(t,t') \wedge (s',t') \in \rho \end{aligned}$$

3. This case is symmetric to case **2.** (see above).

$\Leftarrow$ Let $K_1 \sim K_2$. Then, by Theorem 3.2.3, $K_1 + K_2$ is a common refinement of K_1 and K_2.

□

Intuitively, $s \sim t$ iff the values of all propositions in these states are consistent (condition 1), and s and t have consistent successors. The latter means that for every definite, i.e., *must*, successor s' of s, there exists some possible, i.e., *may*, successor t' of t where s' and t' are consistent (condition 2), and for every definite, i.e., must, successor t' of t, there is some possible, i.e., may, successor s' of s, consistent with t' (condition 3). To prove

that K_1 and K_2 are consistent, we simply need to match *every* must transition of one model to some may transition of the other. It is not necessary to match may transitions – they can either evolve to must or be removed without causing inconsistency.

For example, $\mathbf{CM_1}'$ and $\mathbf{CM_2}'$ in Figures 3.2(a)-(b) are consistent with the consistency relation $\{(s_0, t_0), (s_1, t_1), (s_2, t_2), (s_2, t_3)\}$. On the other hand, $\mathbf{CM_2}'$ and $\mathbf{CM_3}$ in Figures 3.2(b) and (d) are inconsistent: t_1, the successor of t_0, disagrees with r_2 the successor of r_0 on the value of proposition `fo`. Since r_2, a must successor of r_0, cannot be matched to any successor of t_0, t_0 and r_0 are inconsistent, and thus so are $\mathbf{CM_2}'$ and $\mathbf{CM_3}$.

In our example in Figure 3.2, there is a unique consistency relation between models $\mathbf{CM_1}'$ and $\mathbf{CM_2}'$. However, in general, the consistency relation between a consistent pair of KMTSs K_1 and K_2 is not unique. Larsen et. al. (Larsen *et al.*, 1995) formalized the notion of *independence* between partial models (with $R^{must} \subseteq R^{may}$) and showed that there always exists a unique consistency relation for a pair of independent partial models. Models $\mathbf{CM_1}'$ and $\mathbf{CM_2}'$ in Figure 3.2 satisfy the independence conditions of (Larsen *et al.*, 1995), and hence, have a unique consistency relation.

If there exists a consistency relation $\sim$ over the states of K_1 and K_2, the construction of a merged model is straightforward: Every pair of consistent states is merged by computing the truth-disjunction, i.e., $\sqcup$ (see Section 2.1.1), of their corresponding propositions to form a single state in the combined model. For example, the merge of a pair s and t of consistent states such that $L(s, p) = \mathsf{m}$ and $L(t, p) = \mathsf{f}$ is state (s, t) at which p is f. We put a *must* transition between any pair (s, t) and (s', t') of states in the merged model, if there is at least a *must* transition between s and s' and a *may* transition between t and t', or vice versa. Finally, we put a *may* transition between any pair (s, t) and (s', t'), if there is at least one *may* transition between s and s' and one *may* transition between t and t'.

Definition 3.2.5 $(K_1 + K_2)$ *Let K_1 and K_2 be KMTSs, and let $K_1 \sim K_2$. We define a*

merge *of* K_1 *and* K_2*, denoted* $K_1{+}K_2$*, as a tuple* $(S_+, (s_0, t_0), R_+^{must}, R_+^{may}, L_+, AP_1 \cup AP_2)$*, where*

1. $S_+ = \{(s,t) \mid s \sim t\}$
2. $R_+^{must} = \{((s,t),(s',t')) \mid (R_1^{must}(s,s') \wedge R_2^{may}(t,t')) \vee (R_1^{may}(s,s') \wedge R_2^{must}(t,t'))\}$
3. $R_+^{may} = \{((s,t),(s',t')) \mid R_1^{may}(s,s') \wedge R_2^{may}(t,t')\}$
4. $\forall p \in AP_u \cdot L_+((s,t),p) = L_1(s,p) \sqcup L_2(t,p)$

$K_1 + K_2$ is a fragment of the cross-product of K_1 and K_2: its state-pace only includes tuples (s,t) such that s and t are consistent. Note that by our construction in Definition 3.2.5, $K_1 + K_2$ is a KMTS, i.e., its *must* transitions is a subset of its *may* transitions. This is because the condition for *may* transitions, i.e., condition 3, is weaker than that for *must* transitions, i.e., condition 2.

Theorem 3.2.3 *Let* K_1 *and* K_2 *be partial consistent models. Then,* $K_1 + K_2$ *is their common refinement.*

Before we give the proof of the above theorem, we provide an inductive definition, equivalent to Definition 2.3.2, for the refinement relation $\preceq$.

Definition 3.2.6 *We define a sequence of refinement relations* $\preceq^0$, $\preceq^1$, ... *on* $S_1 \times S_2$ *as follows:*

- $s \preceq^0 t$ *iff* $L_1(s,p) \preceq L_2(t,p)$ *for all* $p \in AP_u$*, and*
- $s \preceq^{n+1} t$ *iff*

1. $\forall p \in AP_u \cdot L_1(s,p) \preceq L_2(t,p)$
2. $\forall s' \in S_1 \cdot R_1^{must}(s,s') \;\Rightarrow\; \exists t' \in S_2 \cdot R_2^{may}(t,t') \;\wedge\; s' \preceq^n t'$
3. $\forall t' \in S_2 \cdot R_2^{must}(t,t') \;\Rightarrow\; \exists s' \in S_1 \cdot R_1^{may}(s,s') \;\wedge\; s' \preceq^n t'$

We say $s \preceq t$ *iff* $s \preceq^i t$, *for all* $i \geq 0$.

Note that since K_1 and K_2 are finite structures, the sequence $\preceq^0$, $\preceq^1$, ... is finite as well.

Proof:

We proceed in two steps:

I. We first show that for every $s \in S_1$ if $s \sim t$, then $s \preceq (s,t)$. It suffices to show $s \sim t \Rightarrow s \preceq^i (s,t)$ for all $i \geq 0$. We prove it by induction on i:

Base case. $s \sim t \Rightarrow s \preceq^0 (s,t)$.

$$
\begin{aligned}
& s \preceq^0 (s,t) \\
\Leftrightarrow\ & \text{(by the definition of } \preceq^0\text{)} \\
& \forall p \in AP_u \cdot L_1(s,p) \preceq L_+((s,t),p) \\
\Leftrightarrow\ & \text{(since } L_+((s,t),p) = L_1(s,p) \sqcup L_2(t,p)\text{)} \\
& \forall p \in AP_u \cdot L_1(s,p) \preceq L_1(s,p) \sqcup L_2(t,p)) \\
\Leftrightarrow\ & \text{(by the properties of } \sqcup\text{)} \\
& \text{true}
\end{aligned}
$$

Inductive case. Suppose $s \sim t \Rightarrow s \preceq^n (s,t)$. We prove that

$$s \sim t \Rightarrow s \preceq^{n+1} (s,t)$$

By Definition 3.2.6, we need to show:

1. $L_1(s,p) \preceq L_+((s,t),p)$

2. $\forall s' \in S_1 \cdot R_1^{must}(s,s') \Rightarrow \exists (s',t') \in S_1 \times S_2 \cdot$
$R_+^{must}((s,t),(s',t')) \wedge s' \preceq^n (s',t')$

3. $\forall (s',t') \in S_1 \times S_2 \cdot R_+^{must}((s,t),(s',t')) \Rightarrow$
$\exists s' \in S_1 \cdot R_1^{must}(s,s') \wedge s' \preceq^n (s',t')$

1. From $L_+((s,t),p) = L_1(s,p) \sqcup L_2(t,p)$, and $L_1(s,p) \preceq L_1(s,p) \sqcup L_2(t,p)$

2.

$$\begin{aligned}
& \forall s' \in S_1 \cdot R_1^{must}(s, s') \\
\Rightarrow\ & \text{(since } s \sim t \text{ and by Definition 3.2.4, condition 2)} \\
& R_1^{must}(s, s') \wedge \exists t' \in S_2 \cdot R_2^{must}(t, t') \wedge s' \sim t' \\
\Rightarrow\ & \text{(by the properties of } \sqcup) \\
& \exists t' \in S_2 \cdot R_1^{must}(s, s') \vee R_2^{must}(t, t') \wedge s' \sim t' \\
\Rightarrow\ & \text{(since } R_+^{must}((s,t),(s',t')) \iff R_1^{must}(s, s') \vee R_2^{must}(t, t')) \\
& \exists (s', t') \in S_1 \times S_2 \cdot R_+^{must}((s,t),(s',t')) \wedge s' \sim t' \\
\Rightarrow\ & \text{(by the inductive hypothesis)} \\
& \exists (s', t') \in S_1 \times S_2 \cdot R_+^{must}((s,t),(s',t')) \wedge s' \preceq^n (s', t') \\
\Rightarrow\ & \text{Since } K_1 + K_2 \text{ is a MTS} \\
& \exists (s', t') \in S_1 \times S_2 \cdot R_+^{may}((s,t),(s',t')) \wedge s' \preceq^n (s', t')
\end{aligned}$$

3.

$$\begin{aligned}
& \forall (s', t') \in S_1 \times S_2 \cdot R_+^{must}((s,t),(s',t')) \\
\Rightarrow\ & \text{(by the definition of } R_+) \\
& (R_1^{must}(s, s') \vee R_2^{must}(t, t')) \wedge s' \sim t' \\
\Rightarrow\ & \text{(by } \vee \text{ properties)} \\
& \exists s' \in S_1 \cdot R_1^{must}(s, s') \wedge s' \sim t' \\
\Rightarrow\ & \text{(by the inductive hypothesis)} \\
& \exists s' \in S_1 \cdot R_1^{must}(s, s') \wedge s' \preceq^n (s', t') \\
\Rightarrow\ & \text{Since } K_1 \text{ is a MTS} \\
& \exists s' \in S_1 \cdot R_1^{may}(s, s') \wedge s' \preceq^n (s', t')
\end{aligned}$$

II. Similarly, we show that for every $t \in S_2$, if $s \sim t$, then $t \preceq (s, t)$.

Since K_1 and K_2 are consistent, we have $s_0 \sim t_0$. By **I.** and **II.**, we obtain $s_0 \preceq (s_0, t_0)$ and $t_0 \preceq (s_0, t_0)$. This implies that $K_1 \preceq K_1 + K_2$ and $K_2 \preceq K_1 + K_2$. Therefore, $K_1 + K_2$ is a common refinement of K_1 and K_2. □

For example, model $\mathbf{CM_4}$ in Figure 3.2(e) is the merge of $\mathbf{CM_1}'$ and $\mathbf{CM_2}'$ in Figures 3.2(a) and (b) respectively, and hence, the merge of $\mathbf{CM_1}$ and $\mathbf{CM_2}$ in Figures 3.1(a) and (b), respectively. The consistency relation between $\mathbf{CM_1}$ and $\mathbf{CM_2}$ is $\{(s_0, t_0), (s_1, t_1), (s_2, t_2), (s_2, t_3)\}$. For this example, $\mathbf{CM_4}$ happens to be the most precise merge, i.e., the least common refinement, but this is not necessarily the case in general. $\mathbf{CM_4}$ provides a more complete description of the camera's photo-taking function: it describes the behaviour of the shutter and the built-in flash of the camera as well as its focusing feature. It can be seen that the properties in Table 3.1 all hold over $\mathbf{CM_4}$.

We conclude this section by discussing the complexity of deciding consistency and computing the merged model. The consistency relation in Definition 3.2.4 can be characterized as a bisimulation game (Stirling, 1999) where one player (say Player I) tries to show that models K_1 and K_2 are inconsistent and Player II tries to prove the opposite. In each move, Player I chooses a must transition in either K_1 or K_2 and Player II should respond by finding a may transition in the other model that matches the choice of Player I. Player I wins, i.e., models are inconsistent, if it can find a must transition that Player II fails to match. Player II wins, i.e., models are consistent, if the play is infinite, or if the play reaches a position where Player I has no transition to choose.

A complete formulation of the bisimulation game is available in (Stirling, 1999). The game described above, i.e., the consistency game, is essentially the same as the bisimulation game except that Players in bisimulation games move on classical Kripke structures, and have only one transition type to choose. But, in the consistency game, Player I only chooses must transitions and Player II – may transitions. Deciding bisimulation games is PTIME-complete (Balcázar *et al.*, 1992), and so is deciding a consistency relation between two partial models. Computing a common refinement for a given consistency relation can be done in linear time (see Definition 3.2.5). Thus, merging partial Kripke structures is PTIME-complete.

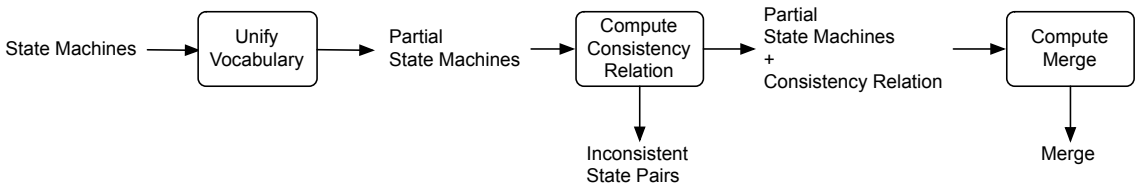


Figure 3.3: An overview of the tool support

In (Hussain & Huth, 2004), it is shown that the complexity of computing a common refinement of a set of partial models is PTIME-complete if the cardinality of the set is *fixed*, and in (Antonik *et al.*, 2008), it is shown that computing a common refinement of a *finite but arbitrary large* set of partial models is PSPACE-hard.

3.3 Tool Support

We have developed a proof of concept implementation of our approach to merging partial and consistent state machines discussed in Section 3.2. Our prototype has three key steps shown in Figure 3.3: In the first step, we extend the vocabulary of each input state machine to the unified set of propositions, i.e., AP_u. The resulting models, which are expressed as partial Kripke structures, are sent to the next step where a consistency relation is computed between models. If the consistency relation does not exist, the process returns a set of state pairs that are inconsistent, and then, terminates. Otherwise, if the consistency relation exits, the merged model is computed in the third step and is returned to the user.

Unifying the vocabulary of state machines and merging models with respect to a given consistency relation, i.e., steps one and three in Figure 3.3, are straightforward. Here, we discuss our algorithm for the second step of the process in Figure 3.3. This algorithm, which is shown in Figure 3.4, uses a greatest fixpoint computation for finding a consistency relation between two models. This algorithm is essentially similar the standard algorithms for computing simulation and bisimulation relations (Milner, 1989).

It first starts with the largest possible consistency relation, i.e., $\rho = \Sigma_1 \times \Sigma_2$ (line 1). It then iteratively refines ρ by removing any inconsistent pair (s, t) of states from ρ (lines 2-14). This fixpoint computation terminates when there are no inconsistent pairs (s, t) in ρ. The resulting ρ is a consistency relation if it includes the pair (s_0, t_0) (line 16). Otherwise, the input models are inconsistent and the algorithm returns the set of inconsistent states in K_1 and K_2 (line 18). The complexity of the algorithm in Figure 3.4 is $O(m^2 \times n^2)$, where n and m are the maximum number of transitions of K_1 and K_2, respectively.

3.4 Related Work

Our approach to model merging presented in this chapter has two main characteristics: (1) It is defined for KMTSs, and (2) It is defined based on the mathematical notion of common refinement. In Chapter 4, we present a more general merging procedure that can handle models with behavioural inconsistencies. The mathematical basis of both of these two merge procedures is common refinement; however, they are defined over two different (non-classical) modelling formalisms: KMTSs in this chapter, and parameterized state machines in Chapter 4. In this section, we survey the related work on non-classical modelling formalisms and defer the detailed comparison with the research on model merging to Section 4.9 after presenting a broader approach to merging in Chapter 4.

The use of non-classical models such as partial models has been motivated from two different perspectives in the software engineering literature: The first motivation comes from the software refinement paradigm. In this paradigm, a specification model is stepwise refined by well-defined design transformations into a final implementation model. Partial models have been advocated in this context as a way to enable specification and analysis of software systems at early stages of development when there is not enough information about the system-to-be at hand, or when modellers simply want to defer

Algorithm. ConsistencyRelation

Input: Partial models K_1 and K_2 with state spaces Σ_1 and Σ_2.

Output: A consistency relation $\rho \subseteq \Sigma_1 \times \Sigma_2$.

1: $\rho = \Sigma_1 \times \Sigma_2$
2: **do**
3: $\mathsf{changed} = false$
4: **for** every $(s,t) \in \rho$ **:**
5: **if** s and t disagree on the value of some proposition $p \in AP_u$ **:**
6: $\rho = \rho \setminus \{(s,t)\}$ *// s and t are inconsistent*
7: $\mathsf{changed} = true$
8: **if** $\exists s' \cdot R_1^{must}(s,s') \wedge \forall t' \cdot R_2^{may}(t,t') \Rightarrow (s',t') \notin \rho$ **:**
9: $\rho = \rho \setminus \{(s,t)\}$ *// s and t are inconsistent*
10: $\mathsf{changed} = true$
11: **if** $\exists t' \cdot R_2^{must}(t,t') \wedge \forall s' \cdot R_1^{may}(s,s') \Rightarrow (s',t') \notin \rho$ **:**
12: $\rho = \rho \setminus \{(s,t)\}$ *// s and t are inconsistent*
13: $\mathsf{changed} = true$
14: **while** changed **:**
15: **if** $(s_0,t_0) \in \rho$ **:**
16: **return** ρ *// consistency relation is computed*
17: **else**
18: **return** $\Sigma_1 \times \Sigma_2 \setminus \rho$ *// K_1 and K_2 are inconsistent*

Figure 3.4: Algorithm for computing a consistency relation.

decisions about some system aspects to future refinements (Larsen & Thomsen, 1988; Larsen, 1989; Huth *et al.*, 2001). The second perspective is that of software abstraction

which is the reverse of refinement. Abstraction is the process of building an approximation of a concrete system that is smaller than the original system, but yet enables conclusive analysis of properties of interest (Clarke *et al.*, 1994). Ideally, abstract models should be small to allow scalable analysis, and should be expressive enough to enable reasoning about a large set of properties. Partial formalisms seem to be a suitable choice because they provide more expressiveness than classical formalisms without causing a significant increase in the size (Dams *et al.*, 1997).

We distinguish three different groups of non-classical modelling formalisms: The first is *Partial Kripke Structures (PKSs)* (Bruns & Godefroid, 2000), and its equivalent formalisms, namely, *Kripke Modal Transition Systems (KMTSs)* (Huth *et al.*, 2001), and *Modal Transition Systems (MTSs)* (Larsen & Thomsen, 1988). PKSs are Kripke structures with 3-valued propositions. KMTSs are PKSs with *must* and *may* transitions such that *must* $\subseteq$ *may*. We used this formalism in this chapter to describe partial models. MTSs are LTSs with *must* and *may* transitions constrained by the condition that *must* $\subseteq$ *may*. Formalisms in this group can be encoded as 3-valued Kripke structures (Chechik *et al.*, 2003). The second group is *Mixed Transition Systems (MixTSs)* (Dams *et al.*, 1997; Cleaveland *et al.*, 1995). These are LTSs with *must* and *may* transitions that do not put any restrictions on the relationship between *may* and *must*, and hence, extend MTSs. MixTSs can be encoded as 4-valued (i.e., Belnap (Belnap, 1977)) Kripke structures (Gurfinkel & Chechik, 2006). The third group is *Hyper Transition Systems (HTSs)* (Shoham & Grumberg, 2004; de Alfaro *et al.*, 2004b; Shoham & Grumberg, 2006), that are MixTSs where *must* transitions can be *hyper-transitions*, i.e., *must* transitions may lead into sets of states rather than single states.

All the formalisms in these three groups can preserve arbitrary temporal properties, e.g., full μ-calculus, L_μ, (Kozen, 1983). The mechanism for property preservation over these formalisms is via two simulation relations: One relating *necessary* or *must* behaviours, and the other relating *possible* or *may* behaviours. The former simulation

preserves existential properties and the latter universal ones.

It is known that formalisms in the first and second group can be translated to HTSs (de Alfaro *et al.*, 2004b), and HTSs are subsumed by tree automata (Dams & Namjoshi, 2005). In (Wei *et al.*, 2008), it is shown that models in the three groups are equally expressive when they are constructed based on the framework of abstract interpretation (Cousot & Cousot, 1977).

These formalisms have been studied and used in the context of software abstraction (e.g., (Dams *et al.*, 1997; Shoham & Grumberg, 2004)). In the context of model-based development, MTSs have been used as the target formalism for synthesis when the requirements are given as a combination of scenarios and safety properties (Uchitel *et al.*, 2007), or when the scenarios are existential properties (Sibay *et al.*, 2008). In this chapter, we used KMTSs to formalize partial perspectives on a single feature of a system where perspectives use different sets of vocabulary, and in the next chapter, we use parameterized state machines, inspired by MixTSs, to formalize the merge of models with inconsistent behaviours.

3.5 Conclusion

In this chapter, we described a technique for merging partial descriptions of the system behaviour. We chose KMTSs to formalize partial models and provided a merge procedure that automatically determines whether two partial models are consistent, and if so, computes their merge. We showed that our merge procedure preserves temporal properties of the input models in their merge. We also reported on a prototype implementation of our approach.

The major limitation of the work discussed in this chapter is lack of support for handling inconsistent models. This limitation renders this work inapplicable to many realistic case studies. The consistency relation defined in Definition 3.2.4 simply does not

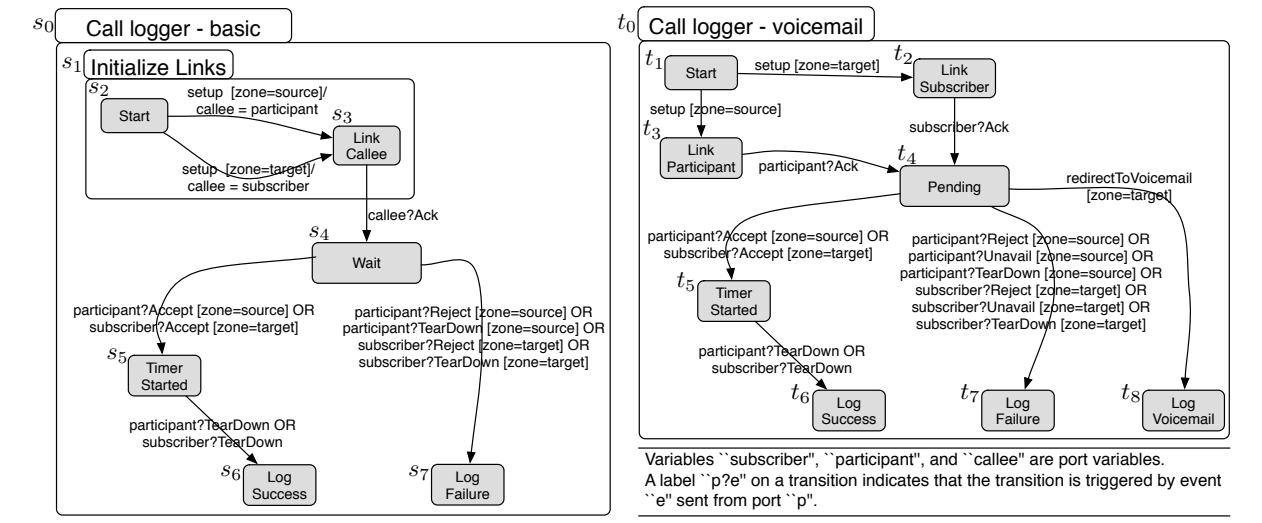
exist between models with inconsistent behaviours. Therefore, it is not clear how such models should be related to one another. In the next chapter, we address this limitation by providing an approach for merging variant and potentially inconsistent descriptions of the system behaviour. The key to making merging of inconsistent models possible is to approximate the relation between models using a quantified notion of behavioural similarity instead of trying to compute an exact consistency relation between them. The resulting relation can be used to build a common refinement in a similar way a common refinement is computed for consistent models (see Definition 3.2.5). The difference is that to be able to preserve behavioural inconsistencies between variant models, a more expressive formalism should be used for describing the merged model. We discuss the details of our technique for merging variant models in the next chapter.

Chapter 4

Merging Variant Feature Specifications

4.1 Introduction

In this chapter, we present an approach for merging variant models of an individual feature with the goal of capturing commonalities and variabilities between the variants and facilitating their maintenance. Our merging approach is defined for Statecharts models and exploits both structural and semantic information in the models, and ensures that behavioural properties are preserved. Our merge is grounded on two model management operators (Brunet *et al.*, 2006): *Match* for finding relationships between models, and *Merge* for combining models with respect to known relationships between them. Our Match operator includes heuristics for finding terminological, structural, and semantic similarities between models. Our Merge operator parameterizes variabilities between the input models so that their behavioural properties are guaranteed to hold in their merge. We illustrate and evaluate our work by applying our operators to a set of AT&T telecommunication features described as Statecharts.



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

Figure 4.1: Simplified variants of the call logger feature.

4.1.1 Motivating Example

Domain. We motivate our work with a scenario for maintaining variant feature specifications at AT&T. These executable specifications are modules within the Distributed Feature Composition (DFC) architecture (Jackson & Zave, 1998), and form part of a consumer voice-over-IP service. In the current implementation of DFC (Bond *et al.*, 2004), the features are written using Statecharts.

One feature of the voice-over-IP service is "call logging", which makes an external record of the disposition of a call allowing customers to later view information on calls they placed or received. At an abstract level, the feature works as follows: It first tries to setup a connection between the caller and the callee. If for any reason (e.g., caller hanging up or callee not responding), a connection is not established, a failure is logged; otherwise, when the call is completed, information about the call is logged.

Initially, the functionality was designed only for basic phone calls, for which logging

is limited to the direction of a call, the address location where a call is answered, success or failure, and the duration if it succeeds. Later, a variant of this feature was developed for customers who subscribe to the voicemail service. Incoming calls for these customers may be redirected to a voicemail resource, and hence, the log information should include the voicemail status as well. Figure 4.1 shows *simplified* views of the *basic* and *voicemail* variants of this feature. To avoid clutter, we combine transitions that have the same source and target states using disjunction (OR).

In the DFC architecture, telecom features may come in several variants to accommodate different customers' needs. The development of these variants is often distributed across time and over different teams of people, resulting in the construction of independent but *overlapping* models for each feature. For example, the behaviour "After a connection is set up, a successful call will be logged if the subscriber or the participant sends Accept" holds in both models in Figure 4.1 (through paths from s_4 to s_6, and t_4 to t_6 in basic and voicemail, respectively). This behaviour is a potential overlap between these models.

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

4.1.2 Contributions of This Chapter

Match and Merge are recurring problems arising in different contexts. Our motivating example illustrates one of the many applications of these operators. Implementation of Match and Merge involves answering several questions. Particularly,

- What criteria should we use for identifying correspondences between different models?

- How can we quantify these criteria?
- How can we construct a merge given a set of models and their correspondences?
- How can we distinguish between shared and non-shared parts of the input models in their merge?
- What properties of the input models should be preserved by their merge?

In this chapter, we address these questions for the Statecharts notation. The contributions of this chapter are as follows:

- A versatile Match operator for Statecharts (Section 4.5). Our Match operator uses a range of heuristics including typographic and linguistic similarities between the vocabularies of different models, structural similarities between the hierarchical nesting of model elements, and semantic similarities between models based on a quantitative notion of behavioural bisimulation. We apply our Match operator to a set of telecom feature specifications developed by AT&T. Our evaluation indicates that the approach is effective for finding correspondences between real-world Statecharts models (Section 4.8).
- A Merge operator for Statecharts (Section 4.6). We provide a procedure for constructing behaviour-preserving merges that also respect the hierarchical structuring of the input models.

4.1.3 Organization of This Chapter

Section 4.2 provides an overview of our Match and Merge operators. Section 4.3 outlines background and fixes notation. Section 4.4 summarizes our major assumptions on the input models for our Match and Merge operators. Section 4.5 introduces our Match operator, and Section 4.6 – our Merge operator. Section 4.7 describes our tool support, and Section 4.8 – our evaluation. Section 4.9 compares our contributions with related

work. Section 4.10 discusses the limitations of the work presented in this chapter, and Section 4.11 concludes the chapter.

4.2 Overview of Our Approach

In this section, we explain the key ideas in our Match and Merge operators by illustrating them using the models in Figure 4.1.

Our Match operator takes as input two models and generates a set of state pairs as a correspondence relation between input models. The main challenge in devising a usable Match operator is finding a set of effective heuristics that can imitate the reasoning of a domain expert. In our work, we use two types of heuristics: static and behavioural. Static heuristics use semantic-free attributes, such as element names, for measuring similarities. For the models in Figure 4.1, static heuristics would suggest a number of good correspondences, e.g., the pairs (s_6, t_6), and (s_7, t_7); however, these heuristics would miss several others including (s_3, t_3), (s_3, t_2) and (s_4, t_4). These pairs are likely to correspond not because they have similar static characteristics, but because they exhibit similar dynamic behaviours. Our behavioural heuristic can find these pairs.

Our Match operator produces a correspondence relation between states in the two models. For the models of Figure 4.1, it may yield the correspondence relation shown in Figure 4.8(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 4.8(c).

In contrast to matching, merging is not heuristic, and is almost entirely automatable. Given a pair of models and a correspondence relation between them, our Merge operator automatically produces a merge that: (1) preserves the behavioural properties of the input models, (2) respects the hierarchical structure of these models, and (3) distinguishes

between shared and non-shared behaviours of these models by attaching appropriate guard conditions to non-shared transitions. Figure 4.9, shows the merge of the models of Figure 4.1 with respect to the relation in Figure 4.8(c). In the merge, non-shared transitions are guarded by boldface conditions representing the models they originate from.

This merge is behaviour-preserving. For example, the property "After a connection is set up, a successful call will be logged if the subscriber or the participant sends Accept" holds in both models in Figure 4.1, and is thus preserved in their merge as a shared behaviour (denoted by the path from state (s_4, t_4) to (s_6, t_6)). The property "After a connection is set up, a voicemail will be logged if the call is redirected to the voicemail service", which holds over the voicemail variant but not over the basic, is represented as a parameterized behaviour in the merge (denoted by the transition from (s_4, t_4) to t_8), and is preserved only when its guard holds. The merge also respects the hierarchical structure of the input models, providing users with a merge that has the same conceptual structure as the input models.

4.3 Background

In this section, we fix the notation and provide background information on (1) the Statecharts dialect studied in this chapter, (2) our technique for converting Statecharts to flat state machines, and (3) the formalism we use to distinguish between shared and non-shared behaviours between state machine variants.

4.3.1 Statecharts

While our work is general and can be applied to various Statecharts dialects, in this chapter, we ground our discussion on a particular dialect, called ECharts (Bond, 2006). ECharts provides well-defined deterministic semantics for the Statecharts language, and

is suitable for detailed design and implementation. The AT&T telecom features are specified in ECharts.

Definition 4.3.1 (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 a partial order defining the state hierarchy tree (or hierarchy tree, for short);* E *is a finite set of events;* V *is a finite set of variables; and* R *is a finite set of transitions, each of which is of the form* $\langle s, e, c, \alpha, s', prty \rangle$*, 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. The hierarchy tree $<_h$ defines a partial order on states with the top superstate as root and the atomic states as leaves. For example, in the Statecharts for the basic call logger model in Figure 4.1, s_0 is the root, s_2 through s_7 are leaves, and s_1 is neither. The set $\hat{s}$ of initial states is $\{s_0, s_1, s_2\}$. The set E of events is $\{\mathsf{setup}, \mathsf{Ack}, \mathsf{Accept}, \mathsf{Reject}, \mathsf{TearDown}\}$, and the set V of variables is $\{\mathsf{callee}, \mathsf{zone}, \mathsf{participant}, \mathsf{subscriber}\}$. The only actions in Figure 4.1 are callee=participant and callee=subscriber. These actions assign values participant and subscriber to the variable callee, respectively.

ECharts does not permit actions generated by a transition of a Statecharts to trigger other transitions of the same Statecharts (Bond & Goguen, 2002). 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. This simplification enables efficient code generation from this Statecharts dialect (Bond & Goguen, 2002).

This formalism, adapted from (Niu *et al.*, 2003), supports superstates (OR states), but not parallel states (AND states). ECharts uses parallel states with interleaved transition executions (Bond, 2006), and can be translated to the above formalism using the

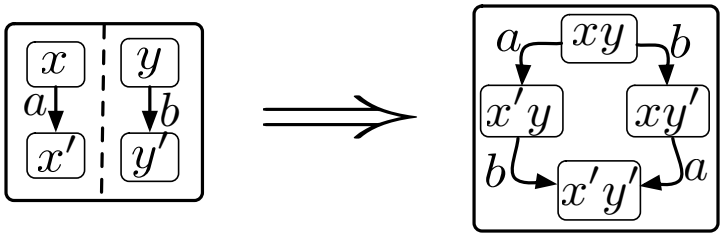


Figure 4.2: Resolving AND-states.

interleaving semantics of (Niu *et al.*, 2003). A simple example of this translation is shown in Figure 4.2. In Statecharts, it may happen that a state and some of its proper descendants both have outgoing transitions with the same event and condition, but different target states. For example, in Figure 4.3(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 Statecharts non-deterministic because on receipt of the event a it is not clear which of the transitions $s_0 \rightarrow s_2$ and $s_1 \rightarrow s_3$ should be taken. In ECharts, transitions with the same event and condition can be made deterministic by assigning globally-ordered priorities to them (using *prty*). For example, in Figure 4.3(a), it is assumed that the inner transitions have higher priority over the outer transitions, and hence, on receipt of a, the transition from s_1 to s_3 is activated. The models shown in Figure 4.1 are already deterministic, i.e., any external event triggers at most one transition in these models. Thus, no prioritization is required for these models.

4.3.2 Flattening

Flattening is known as the process of removing hierarchy in Statecharts models. Flattening is used for several purposes such as constructing operational semantics for Statecharts to model-checking and automatic testing. Our merge procedure described in Section 4.6 is defined over hierarchical state machines, and hence, no flattening is required prior to

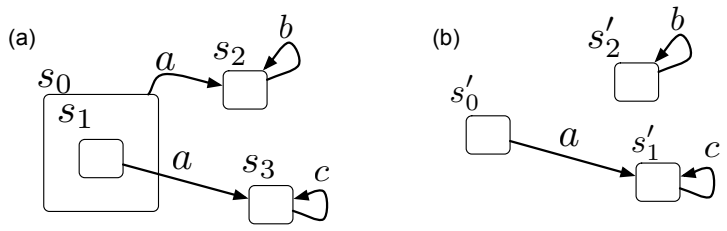


Figure 4.3: (a) Prioritizing transitions to eliminate non-determinism: Transition $s_1 \rightarrow s_3$ has higher priority than transition $s_0 \rightarrow s_2$, and (b) the flattened form of the Statecharts in (a).

the application of merge. The semantics of our merge procedure, however, is defined over flattened Statecharts models, i.e., Labelled Transition Systems (LTSs) (see Definition 2.1.3), and therefore, we need to formally describe how hierarchical Statecharts models are converted to flat state machines. To flatten Statecharts, we first translate them to an intermediate state machine formalism given in Definition 4.3.2, and then discuss how this formalism can be converted to LTSs.

Definition 4.3.2 (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,* E *is a set of input events, and* Act *is a set of output actions.*

Definition 4.3.3 (Flattening) *Let* $M = (S, \hat{s}, <_h, E, V, R)$ *be a Statecharts. 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:

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

$$
\begin{aligned}
E' \quad &= \{e \mid \exists \langle s, e', c, \alpha, s'\rangle \in R \cdot e = e'[c]\} \\
Act' &= \{a \mid \exists \langle s, e', c, \alpha, s'\rangle \in R \cdot a \textit{ appears in the sequence } \alpha)\}
\end{aligned}
$$

State machines in Definition 4.3.2 are similar to LTSs (see Definition 2.1.3) except that in state machines, transitions are labelled by (e, α) where e is an input event and α is a sequence of output actions. In contrast, in LTSs, 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 chapter, we assume that the result of Statecharts flattening is an LTS, i.e., 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 copy of a, i.e., $a?$ for input and $a!$ for output, in the vocabulry of the LTS corresponding to M.

To flatten a hierarchical state machine M, we remove the super-states of M and push the outgoing and incoming transitions of the super-states down to their atomic sub-states. We also remove guards and assume that they are part of events. Note that in Definition 4.3.3, we resolve the priorities between Statecharts transitions, and thus, the transitions of the resulting LTS are no longer prioritized. For example, the LTS in Figure 4.4(b) is the flattened form of the Statecharts in Figure 4.4(a). Similarly, LTSs corresponding to the Statecharts in Figure 4.1 are shown in Figure 4.5. The LTS

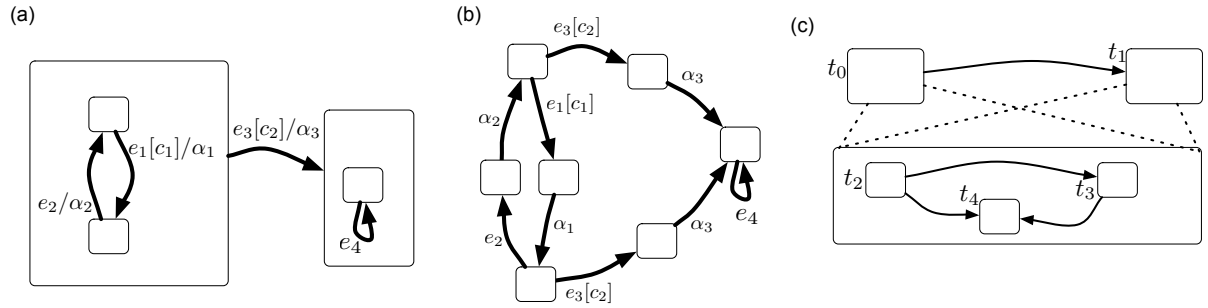


Figure 4.4: Statecharts flattening: (a) An example Statecharts, (b) flattened state machine equivalent to the Statecharts in (a), and (c) an example Statecharts whose superstates share the same sub-states.

corresponding to the Statecharts in Figure 4.3(a) is shown in Figure 4.3(b), illustrating how we resolve priorities during flattening.

Obviously, flattening increases the number of transitions. In situations where superstates share the same sub-states (see Figure 4.4(c) for an example), flattening also increases the number 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 (Alur *et al.*, 1999).

4.3.3 Mixed LTSs

Individual variant models such as those shown in Figure 4.5 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

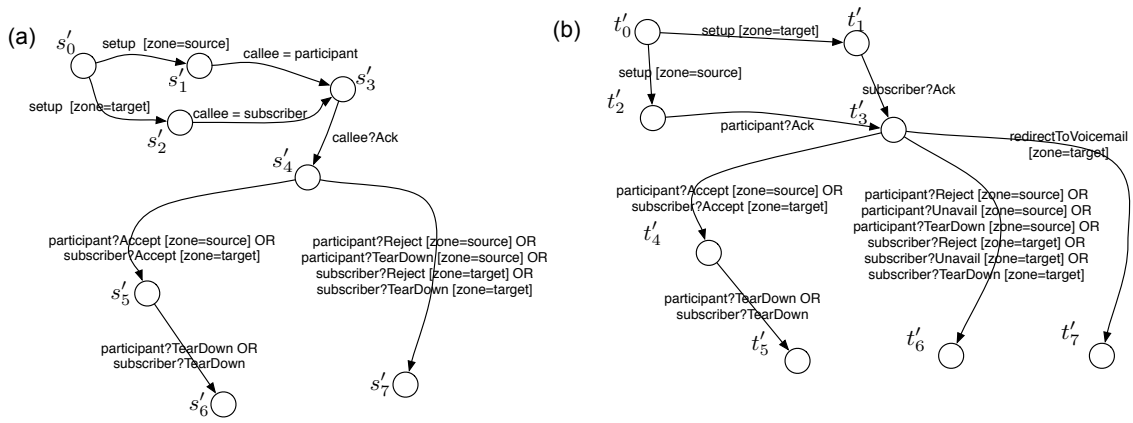


Figure 4.5: LTSs generated by flattening the Statecharts in Figure 4.1.

of behaviours are referred to as *commonalities*, and the latter as *variabilities* (Gomaa, 2004). 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 4.3.4 (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 both 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 (see Definition 2.1.4). Both Mixed LTSs and MixTSs are LTSs with two types of transitions. However, the transition types are used for different purposes in each of these formalisms: In MixTSs, transition types are used to explicitly model possible and required behaviours of a system, whereas in Mixed LTSs, we use transition types to differentiate between shared and non-shared behaviours between model variants.

We define a notion of refinement to formalize the relationship between Mixed LTSs based on the degree of behavioural variabilities that they can capture. Our notion of refinement is very similar to that given in Definition 2.3.4 over MixTSs. The difference is that the refinement in Definition 2.3.4 captures the "more defined than" relation between two partial models, whereas based on our refinement defined below, a model is more refined if it can capture more behavioural variability. For states s and s' of a Mixed LTS L, we write $s \stackrel{e}{\Longrightarrow}^{shared} s'$ to denote $s \stackrel{e}{\Longrightarrow} s'$ in L^{shared}, $s \stackrel{e}{\Longrightarrow}^{nonshared} s'$ to denote $s \stackrel{e}{\Longrightarrow} s'$ in $L^{nonshared}$, and $s \stackrel{e}{\Longrightarrow}^{all} s'$ to denote $s \stackrel{e}{\Longrightarrow} s'$ in L^{all}.

Definition 4.3.5 (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. $\forall s' \in S_1 \cdot \forall e \in E_1 \cup \{\tau\} \cdot s \stackrel{e}{\longrightarrow}^{all} s' \Rightarrow \exists t' \in S_2 \cdot t \stackrel{e}{\Longrightarrow}^{all} t' \wedge \rho(s',t')$
2. $\forall t' \in S_2 \cdot \forall e \in E_2 \cup \{\tau\} \cdot t \stackrel{e}{\longrightarrow}^{shared} t' \Rightarrow \exists s' \in S_1 \cdot s \stackrel{e}{\Longrightarrow}^{shared} s' \wedge \rho(s',t')$

We say 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 M_1 and M_2, respectively.

Intuitively, refinement over Mixed LTSs allows one to convert shared behaviours to non-shared behaviours, while preserving all the already identified non-shared behaviours. More specifically, if L_2 refines L_1, then *every* behaviour of L_1 is present in L_2 either as a shared or non-shared behaviour: Shared behaviours of L_1 may turn to 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 4.3.5, 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 that are not included in the vocabulary of L_1.

Figure 4.6 shows a Mixed LTS where shared transitions are shown as solid arrows and non-shared transitions as dashed arrows. It can be shown that the Mixed LTS in Fig-

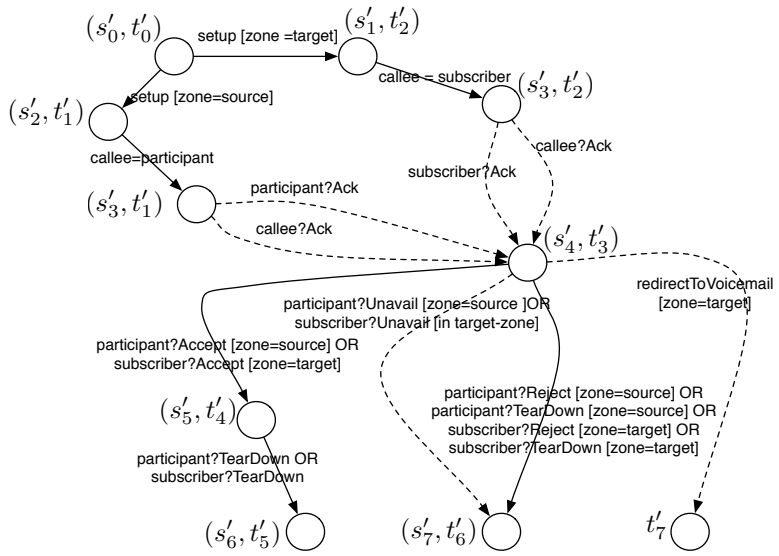


Figure 4.6: Mixed LTS generated by flattening the Statecharts in Figure 4.9: Shared transitions are shown as solid arrows, and non-shared transitions as dashed arrows.

ure 4.6 refines the LTS in Figure 4.5(a) where the relation is $\{(s,(s,x)) \mid s$ and (s,x) are states in Figures 4.5(a) and 4.6, respectively.$\}$.

Theorem 4.3.1 *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 4.3.4, every Mixed LTS L has fragments L^{shared} and L^{all} which are expressible as LTSs. By Definition 4.3.5 and definition of simulation over LTSs given in Definition 2.3.3, for two Mixed LTSs L_1 and L_2 such that L_2 refines L_1, we have L_2^{all} simulates L_1^{all} and L_1^{shared} simulates

L_2^{shared}. By Theorem 2.3.2, we have $\mathcal{L}(L_1^{all}) \subseteq \mathcal{L}(L_2^{all})$ and $\mathcal{L}(L_2^{shared}) \subseteq \mathcal{L}(L_1^{shared})$. More formally,

$L_1 \preceq L_2$

$\Rightarrow$ (By definition of Mixed LTS (Definition 4.3.4), definition of refinement over Mixed LTSs (Definition of 4.3.5) and definition of simulation over LTSs (Definition 2.3.3))

$L_1^{all} \preceq L_2^{all} \wedge L_2^{shared} \preceq L_1^{shared}$

$\Rightarrow$ (By Theorem 2.3.2)

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

□

For example, consider the model in Figure 4.5(a) and its refinement in Figure 4.6. The model in Figure 4.5(a) is an LTS, and hence, its set of non-shared traces is empty. It can be seen that every trace in the model in Figure 4.5(a) is present in the model in Figure 4.6 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 4.6 is present in Figure 4.5(a), i.e., $\mathcal{L}(L_2^{shared}) \subseteq \mathcal{L}(L_1^{shared})$. Finally, the model in Figure 4.6 has some non-shared traces that are not present in the model in Figure 4.5(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 $\mathcal{L}(L_1^{nonshared})$.

4.4 Assumptions

In this section, we describe the major assumptions we make to enable our Match and Merge operators.

We present our operators for Statecharts models $M_1 = (S_1, \hat{s}, <_h^1, E_1, V_1, R_1)$ and $M_2 = (S_2, \hat{t}, <_h^2, E_2, V_2, R_2)$. We assume the sets of events, E_1 and E_2, and environmental/input variables 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 input variables capture observable stimuli, and for these, a unified vocabulary is often developed during upstream lifecycle

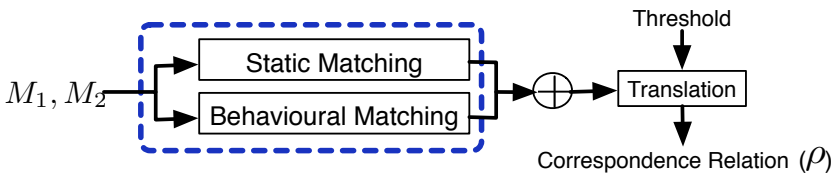


Figure 4.7: Overview of the Match operator.

activities.

Note that since M_1 and M_2 describe variant specifications of the *same* feature, it is unlikely for these models to interact with each other. 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 Statecharts in Figure 4.1 are callee=participant and callee=subscriber. These actions do not cause any interaction between the Statecharts in Figure 4.1. Hence, the Statecharts in Figure 4.1 are non-interacting. Note that studying behavioural models with interacting behaviours and analysing their interactions is the subject of Chapter 5.

4.5 Matching Statecharts

Our Match operator (Figure 4.7) uses a hybrid approach combining static matching (Section 4.5.1) and behavioural matching (Section 4.5.2). Static matching is independent of Statecharts semantics and uses typographic and linguistic similarities between state names, and similarities between state depths in the models' hierarchy trees. Behavioural matching generates similarity degrees between states based on their behavioural semantics. We aggregate these static and behavioural heuristics to produce overall similarity degrees between states (Section 4.5.3). Given a similarity threshold, we can then determine a correspondence relation ρ over the states of the input models (Section 4.5.4).

4.5.1 Static Matching

Typographic Matching assigns to every pair (s, t) of states a normalized value in [0..1] computed by applying the N-gram algorithm (Manning & Schütze, 1999) to the name labels of s and t. Given a pair of strings, this algorithm produces a similarity degree based on counting the number of their identical substrings of length N. We use a generic implementation of this algorithm with trigrams (i.e., N = 3).

Linguistic Matching measures similarity between name labels based on their linguistic correlations, to assign a normalized similarity value to every pair of states. We employ the freely available WordNet::Similarity package (Pedersen *et al.*, 2004) for this purpose.

Depth Matching uses state depths to derive a useful similarity heuristic for models that are at the same level of abstraction. This captures the intuition that states at similar depths are more likely to correspond to each other. Depth matching assigns a normalized value in [0..1] to every pair (s, t) of states. The closer the depths of s and t in their respective hierarchy trees are, the closer this value is to one. Depth matching is not used when the input models are at different levels of abstraction.

4.5.2 Behavioural Matching

Our behavioural matching technique is reminiscent of deciding bisimilarity between state-machines (Milner, 1989). Bisimilarity provides a natural way to characterize behavioural equivalence. Bisimilarity is a recursive notion and can be defined in two ways, forward and backward (Nicola *et al.*, 1990). 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 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 4.1 transit to (forward) bisimilar states s_7 and t_7, respectively, with transitions labelled participant?Reject[zone=source], participant?TearDown[zone=source], subscriber?Reject[zone=target], and subscriber?TearDown[zone=target]. However, despite their intuitive similarity, s_4 and t_4 are dissimilar because their behaviours differ on a few other transitions, e.g., the one labelled redirectToVoicemail[zone=target].

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

In the remainder of this section, we describe the algorithm for the forward case. The backward case is similar. We use the notation $s \xrightarrow{a} s'$ to indicate that s' is a forward neighbour of s. That is, s has a transition to s' labelled a, or s' is child of s where a is a special label called *child*. Treating children states as neighbours allows us to propagate similarities from children to their parents.

Behavioural matching is a total function $\mathcal{B} : S_1 \times S_2 \to [0..1]$. 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 provides a mechanism for users to apply their domain expertise during the

matching process. Since behavioural matching is iterative, user input gets propagated to all tuples and can hence induce an overall improvement in the results of matching. It is also possible to use the results of static matching to initialize the behavioural matching, in the same way that schema matchers can be bootstrapped (Rahm & Bernstein, 2001).

For proper aggregation of similarity degrees between states, our behavioural matching requires a measure for comparing transition labels. A transition label is made up of an event, and optionally, a condition and an action. We compare transition labels using the N-gram algorithm augmented with some simple semantic heuristics. This algorithm is suitable because of the assumption that a shared vocabulary for observable stimuli already exists. We improve transition label comparison by using the variable assignments in the action parts of transition labels for term rewriting. For example, in Figure 4.1, the actions callee = participant and callee = subscriber suggest that the transition label callee?Ack is similar to participant?Ack and subscriber?Ack. We account for this by (automatic) term replacement prior to applying the N-gram algorithm. The algorithm assigns a similarity value $L(a, b)$ in [0..1] to every pair (a, b) of transition labels.

We have also explored the use of analytical reasoning for comparing transition labels. For example, the N-gram algorithm would find a rather small degree of similarity between conditions $(x \wedge y) \vee z$ and $(x \vee z) \wedge (y \vee z)$, whereas analytical reasoning, e.g. by a theorem prover, would identify these conditions as identical. Our experimentation with this idea indicates that such reasoning over labels is expensive and also unnecessary because examples such as the above are not very common in practice.

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

We formalize the computation of $\mathcal{B}^i(s, t)$ as follows. Let $s \xrightarrow{a} s'$. To find the

best match for s' among the forward neighbours of t, we need to maximize the value $L(a,b) \times \mathcal{B}^{i-1}(s',t')$ where $t \xrightarrow{b} t'$.

The similarity degrees between the forward neighbours of s and their best matches among the forward neighbours of t after iteration $i-1$ is computed by $X = \sum_{s \xrightarrow{a} s'} max_{t \xrightarrow{b} t'} L(a,b) \times \mathcal{B}^{i-1}(s',t')$. And the similarity degrees between the forward neighbours of t and their best matches among the forward neighbours of s after iteration $i-1$ are computed by $Y = \sum_{t \xrightarrow{a} t'} max_{s \xrightarrow{b} s'} L(a,b) \times \mathcal{B}^{i-1}(s',t')$. We denote the sum of X and Y by $Sum^i(s,t)$.

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

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

In the above formula, $|succ(s)|$ and $|succ(t)|$ are the number of forward neighbours of s and t, respectively. The larger the $\mathcal{B}^i(s,t)$, the more the behaviours of s and t are alike.

This computation is performed iteratively until the difference between $\mathcal{B}^i(s,t)$ and $\mathcal{B}^{i-1}(s,t)$ for all pairs (s,t) becomes less than a fixed $\varepsilon > 0$. If the computation does not converge, the algorithm stops after some maximum number of iterations.

4.5.3 Combining Different Similarity Measures

To obtain overall similarity degrees between states, we need to combine the results from different heuristics. There are several approaches to this, including linear and nonlinear averages, and machine learning. Learning-based techniques have been shown to be effective when proper training data is available (Mandelin *et al.*, 2006). At this stage, we do not have sufficient training data to employ such techniques. In our current implementation, we use a simple approach based on linear averages.

We generate an aggregate value for static heuristics, denoted by $\mathcal{S}$, by taking the

	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 4.1.

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

(b) A correspondence relation ρ.

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

(c) ρ after revisions of Sec 4.5.4 and sanity checks of Sec 4.6.1.

Figure 4.8: Results of matching for call logger.

maximum of typographic and linguistic similarities, and computing its weighted average with depth similarity. Behavioural similarity, $\mathcal{B}$, is computed as the maximum of forward behavioural and backward behavioural matching. To produce an overall combined measure, denoted $\mathcal{C}$, we take a weighted average of $\mathcal{B}$ with $\mathcal{S}$. Figure 4.8(a) illustrates $\mathcal{C}$ for the models in Figure 4.1. Here, we use a 4-to-1 ratio for averaging name similarities (max. of typographic and linguistic) with depth similarity, and use equal weights for averaging $\mathcal{S}$ and $\mathcal{B}$. These weights, which we arrived at by experimentation, are also used for the evaluation in Section 4.8.

4.5.4 Translating Similarities to Correspondences

To obtain a correspondence relation between M_1 and M_2, the user sets a threshold for translating the overall similarity degrees into a relation ρ. Pairs of states with similarity degrees above the threshold are included in ρ, and the rest are left out. In our example, if we set the threshold value to 60%, we obtain the correspondence relation ρ shown in Figure 4.8(b). Since matching is a heuristic process, ρ should be reviewed and, if

necessary, adjusted by the user. We assume that the user would remove the spurious pairs (s_6, t_7), (s_6, t_8), (s_7, t_6) and (s_7, t_8) from ρ. As we will discuss in Section 4.6.1, the resulting relation needs to be further revised before merge.

4.6 Merging Statecharts

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

4.6.1 Sanity Checks for Correspondence Relations

Before applying the Merge operator, we need to ensure that ρ passes certain sanity checks. To have behaviourally sound merges, the initial states of the input models should correspond. If ρ does not match $\hat{s}$ to $\hat{t}$, we add to the input models new initial states $\hat{s}'$ and $\hat{t}'$ with transitions to the old ones. We then simply add the tuple $(\hat{s}', \hat{t}')$ to ρ. Note that we can lift the behavioural properties of the models with the old initial states to those with the new initial states. For example, instead of evaluating a temporal property p at $\hat{s}$ (resp. $\hat{t}$), we check AXp at $\hat{s}'$ (resp. $\hat{t}'$), where AX denotes the universal next-time operator.

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

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

2. *(relational adequacy)* Either the parent of s is related to an ancestor of t, or the parent of t is related to an ancestor of s by ρ.

Monotonicity ensures that ρ does not relate an ancestor of s to t (resp. t to s) or to a child thereof. Relational adequacy ensures that ρ does not leave parents of both s and t unmapped; otherwise, it would not be clear which state should be the parent of s and t in the merge. Note that descendant, ancestor, parent, and child are all with respect to each model's hierarchy tree, $<_h$.

Pairs in ρ that violate any of the above conditions are reported to the user. In our example, the relation shown in Figure 4.8(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. We assume that the user addresses these conflicts by eliminating (s_1, t_0) and (s_1, t_1) from ρ. The resulting relation, shown in Figure 4.8(c), passes all sanity checks and can be used for merge.

4.6.2 Merge Construction

To merge M_1 and M_2, we first need to identify their shared and non-shared parts with respect to ρ. A state x is *shared* if it is related to some state by ρ, and is *non-shared* otherwise. A transition $r = \langle x, a, c, \alpha, y, prty \rangle$ is *shared* if x and y are respectively related to some x' and y' by ρ, and further, there is a transition r' from x' to y' whose event is a, whose condition is c, whose priority is $prty$, and whose action is α' such that $\alpha = \alpha'$, 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 (Clarke *et al.*, 1999). 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. Notice that there is no requirements for the actions of r and r' to be identical.

The goal of the Merge operator is to construct a model that contains shared behaviours of the input models as normal behaviours and non-shared behaviours as variabilities. To represent variabilities, we use parameterization (Gomaa, 2004): Non-shared transitions

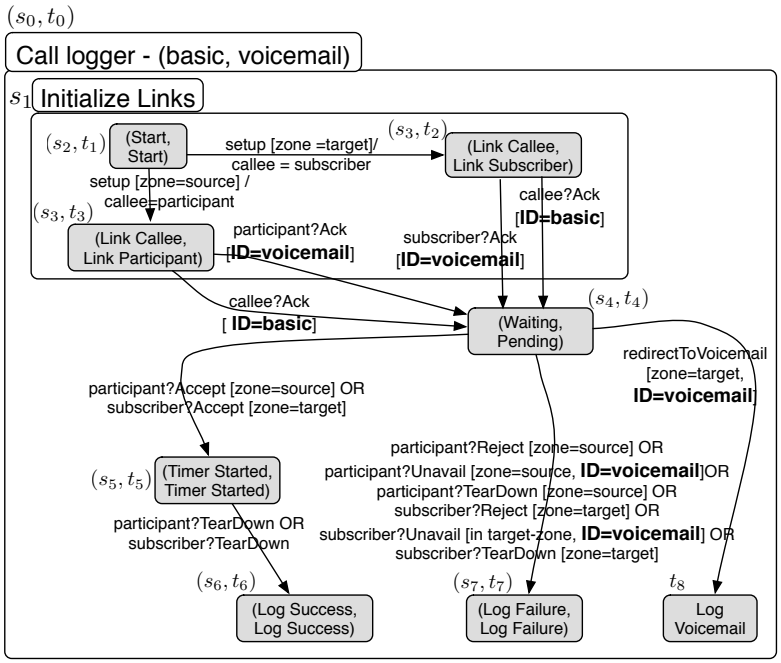


Figure 4.9: Resulting merge for call logger.

are guarded by conditions denoting the transitions' origins, before being lifted to the merge. Non-shared states can be lifted without any provisions – these states are reachable only via non-shared (and hence, guarded) transitions.

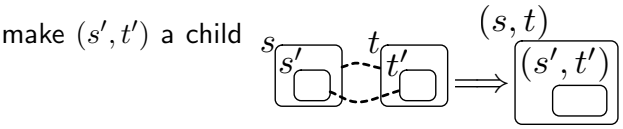
Below, we describe our procedure for constructing a merge. We denote by $M_1 +_\rho M_2 = (S_+, \hat{s}_+, <_h^+, E_+, V_+, R_+)$ the merge of M_1 and M_2 with respect to ρ.

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

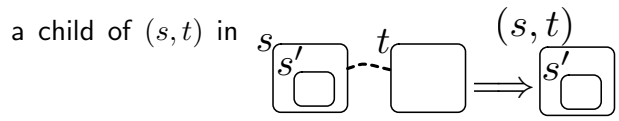
Events and Variables. (E_+ and V_+) The set E_+ of events of $M_1 +_\rho M_2$ is the union of those of M_1 and M_2. The set V_+ of variables of $M_1 +_\rho M_2$ is the union of those of M_1 and M_2 plus a reserved enumerated variable **ID** that accepts values M_1 and M_2.

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

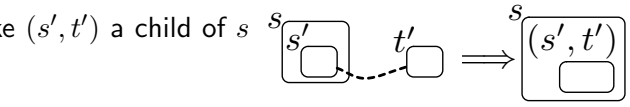
- if s is mapped to t by ρ,
  - if s' is mapped to a child t' of t by ρ, make (s', t') a child of (s, t) in $M_1 +_\rho M_2$.



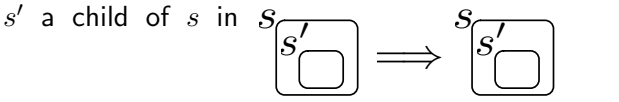
  - otherwise, if s' is non-shared, make s' a child of (s, t) in $M_1 +_\rho M_2$.



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



  - otherwise, if s' is non-shared, make s' a child of s in $M_1 +_\rho M_2$.



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

- **(Shared Transitions)** if r is shared, add to R_+ a transition corresponding to r with event a, condition c, action α (if $\alpha = \alpha'$) or action $\alpha; \alpha'$ (if $\alpha \neq \alpha'$), and priority $prty$. Note that due to the definition of shared transitions and our assumptions in Section 4.4, α and α' are independent, and hence, the order of concatenation of α and α' is unimportant here.
- **(Non-shared Transitions)** otherwise, if r is non-shared, add to R_+ a transition corresponding to r with event a, condition $c \wedge [\mathbf{ID} = \mathbf{M_1}]$, action α, and priority $prty$.

As an example, Figure 4.9 shows the resulting merge for the models of Figure 4.1 with respect to the relation ρ in Figure 4.8(c). The conditions shown in boldface in Figure 4.9 capture the origins of the respective transitions. For example, the transition from (s_4, t_4) to t_8 annotated with the condition **ID=voicemail** indicates a variable behaviour that is applicable only for clients subscribing to voicemail.

Two points need to be noted about our merge construction: (1) The construction requires that states be either atomic or superstates (OR states) – as noted in Section 4.3,

parallel states (AND states) are replaced by their semantically equivalent non-parallel structures before merge. To keep the structure of the merged model as close as possible to the input models, non-shared parallel states can be exempted from this replacement when their descendants are all non-shared too. Such parallel states (and all descendants thereof) can be lifted verbatim to the merge. (2) Our definition of shared transitions is conservative in the sense that it requires such transitions to have identical events, conditions, and priorities in both input models. This is necessary to ensure that merges are behaviourally sound and deterministic. However, such a conservative approach may result in redundant transitions. These redundancies arise due to logical or unstated relationships between the events and conditions used in the input models. For example, in Figure 4.9, the transitions from (s_2, t_1) to (s_3, t_2) and to (s_3, t_3) fire actions callee = subscriber and callee=participant, respectively. Thus, in state (s_3, t_3), the value of callee is equal to participant, and in state (s_3, t_2), it is equal to subscriber. This allows us to replace the event callee?Ack[**ID=basic**] on transition from (s_3, t_2) to (s_4, t_4) by subscriber?Ack[**ID=basic**], and merge the two out-going transitions from (s_3, t_2) into one transition with label subscriber?Ack. Similarly, the two transitions from (s_3, t_3) to (s_4, t_4) can be merged into one transition with label participant?Ack. Identifying such redundancies and addressing them requires human intervention.

4.7 Tool Support

In this section, we describe our implementation for the Match and Merge operators described in Sections 4.5 and 4.6, respectively.

4.7.1 Tool Support for Matching

We have implemented our Match operator and have used it for the evaluation described in Section 4.8. Our Match operator takes Statecharts models stored as XML files and

computes similarity values for static matching, behavioural matching and their combinations. Our implementation also offers the option of generating a binary correspondence relation for a given threshold value. This relationship, after revisions and adjustments by the user, can be used to specify the model mappings in our merge tool described in Section 4.7.2.

Our Match tool is written entirely in Java. It is roughly 4.5K lines of code, of which 1.2K is the N-gram package (Manning & Schütze, 1999), 1K implments the glue code for interacting with WordNet::Similarity package (Pedersen *et al.*, 2004), 1.9K implements our behavioural matching formulation (Section 4.5.2), and the rest is the parser for XML files and the code for interacting with the user. We have made this tool available at `http://www.cs.toronto.edu/~shiva/MatchTool/`.

4.7.2 Tool Support for Merging

We have implemented our Merge operator as part of a model merging tool called TReMer (Tool for Relationship-Driven Model Merging) (Sabetzadeh *et al.*, 2007a). An extension of TReMer, i.e., TReMer+(Sabetzadeh *et al.*, 2008), which is capable of performing global consistency checking is available at `http://www.cs.toronto.edu/~mehrdad/tremer/`. The main idea behind TReMer is to make all information about the relationships between models explicit, and indeed suggest that model relationships should be treated as *first-class artifacts* in the model merging procedure (Brunet *et al.*, 2006). This treatment has two main advantages:

1. It allows us to hypothesize alternative relationships between models and study the result of merge for each alternative. This is particularly useful in distributed development environments where software models are created by developers working in distributed teams, and as a result, one can never be entirely sure how the concepts in different models relate to one another.

2. It provides the flexibility to define different types of model relationships and to associate each type with certain merge algorithms. This addresses the observation that model merging may be utilized to achieve different goals at different stages of the development. For example, in early stages, we may merge a set of models as a way to unify the vocabularies of different stakeholders. For this purpose, it is convenient to treat models as syntactic objects without rigorous semantics. In later stages, however, we often want to account for the semantics of models and produce merges that preserve certain semantic properties. This means that we may need to apply different merge algorithms to a set of models as they move through different stages of development. These algorithms typically require different types of relationships to be built between models (e.g., syntactic and semantic).

TReMer provides a general framework for model merging whereby the relationships between models can be specified explicitly. Figure 4.10 shows an outline of this framework: Given a set of models, users can define different types of relationships between them and for each type, explore alternative ways of mapping the models to one another. The models and a selected set of relationships are then used to compute a merge by applying an appropriate merge algorithm. The merge is then presented to users for further analysis which may lead to the discovery of new relationships or the invalidation of some of the existing ones. Users may then want to start a new iteration by revising the relationships and executing the subsequent activities.

TReMer currently supports two model merging algorithms: One based on our Merge operator described in Section 4.6 for merging behavioural models, and the other for merging conceptual models in requirements modelling. For details on the latter merge algorithm, refer to (Sabetzadeh & Easterbrook, 2006). TReMer consists of two main components: (1) A graphical front-end[1] allowing users to visually express their mod-

[1]The front-end cannot represent hierarchical models. Currently, we use a generalization relation between states to illustrate the state hierarchy tree.

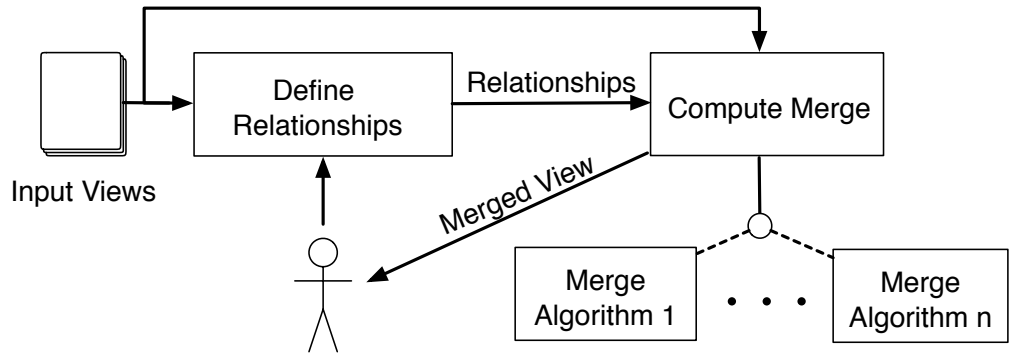


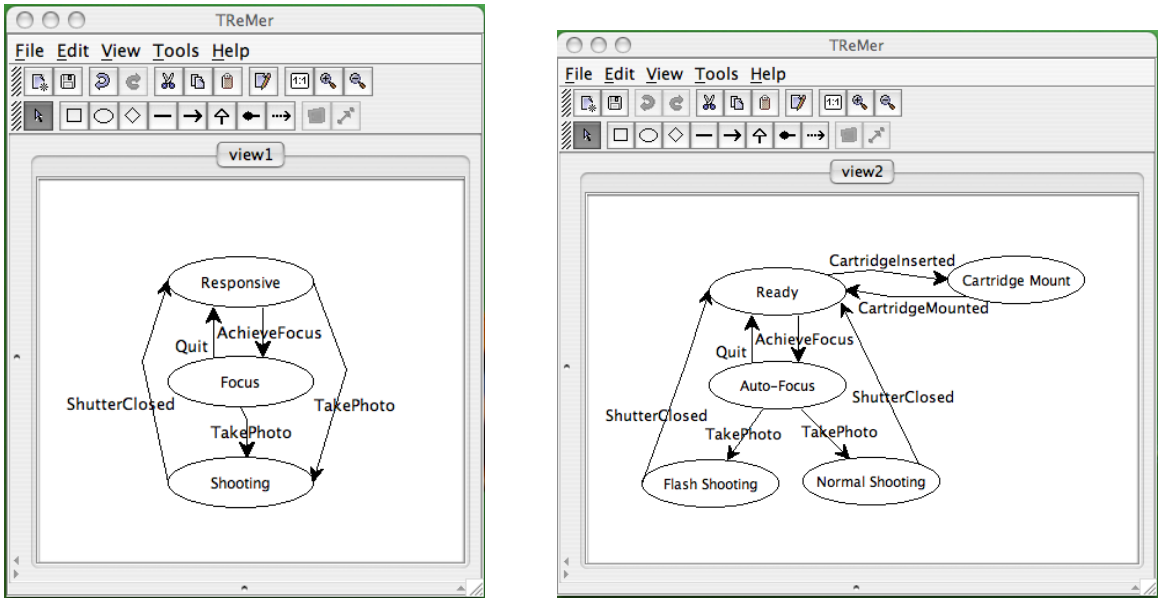
Figure 4.10: Overview of TReMer.

els, specify different relationships between these models, and hypothesize and compute merges, and (2) A library of merge algorithms. These algorithms communicate with the front-end in a uniform way via an abstract interface for the merge operation. The key idea that allows us to have such an interface is the treatment of model relationships as first-class artifacts.

In the rest of this section, we illustrate our behavioural merging tool using a concrete example. We use two views on the photo-taking functionality of a camera[2]: Figure 4.11(a) shows a perspective where the flash and non-flash (normal) shooting modes are not distinguished and the film-cartridge loading procedure is ignored. Further, the perspective has a transition from **Responsive** to **Shooting** meaning that the camera can take a photo even without achieving focus. In the second perspective, Figure 4.11(b), flash and non-flash shooting are distinguished and cartridge loading is modelled.

Behavioural merging of the state-machines in Figure 4.11 proceeds as follows: The user begins by defining a relationship between the states of the input models. Currently, TReMer's user interface allows one to define one-to-one or many-to-one mappings from the states of a model to those of another. To define such mappings, the two models are shown side-by-side. A state correspondence is established by first clicking on a state of the

[2]The camera example is similar to that used in Section 3.1.1 except that here models are *not* behaviourally consistent. Furthermore, in this chapter, we use action-based state machines to describe the camera models, whereas in Chapter 3, we focused on state-based transition systems.



(a) Without Flash (b) With Flash

Figure 4.11: Two perspectives on a camera

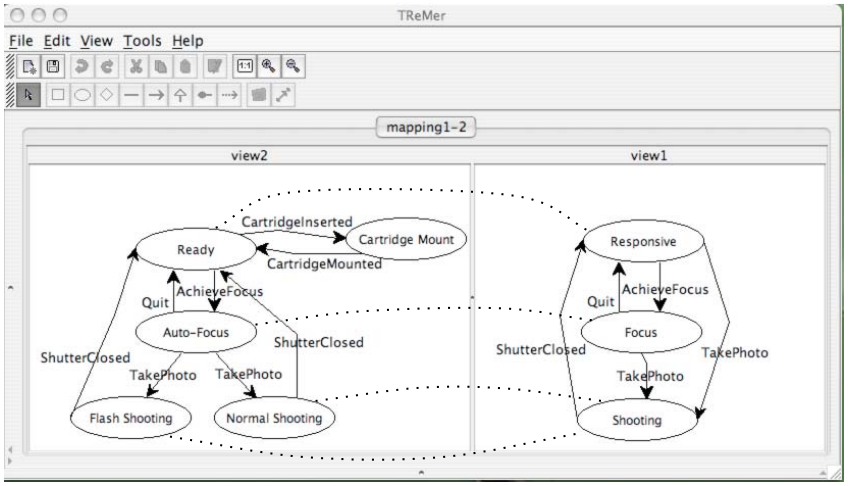


Figure 4.12: Behavioural mapping between the models in Figure 4.11

model on the left and then on a state of the model on the right. For example, Figure 4.12 shows a behavioural mapping between the input state-machines in Figure 4.11. Note that the relationship in Figure 4.12 is a many-to-one relation. To describe many-to-many relations such as the one shown in Figure 4.13 (note the additional mapping between

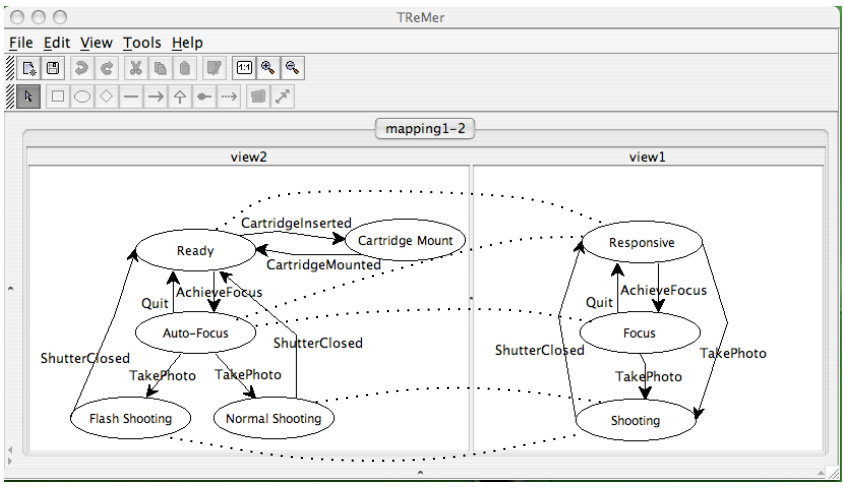


Figure 4.13: A many-to-many relation between the models in Figure 4.11

Auto-Focus and **Responsive**), the user needs to first decompose the relation into two many-to-one mappings, and then specify each mapping separately using the TReMer's user interface.

The next step is to compute the merge with respect to the mappings defined above. TReMer computes the union of the one-to-many mappings between a pair of models to obtain the original many-to-many relation. It then applies the merge procedure described in Section 4.6 to generate a merged model. The result is shown in Figure 4.14. As can be seen from the figure, non-shared behaviours are guarded by conditions denoting the originating model that exhibits those behaviours.

4.8 Evaluation

The ultimate evaluation of our work is whether developers faced with model management tasks find our approach helpful. In some contexts, developers may find it relatively easy to identify matches by hand, for example if the models are small, and the developers are very familiar with them. Our approach to matching is valuable if it offers a quick way to identify appropriate matches with reasonable accuracy, in situations where matches are

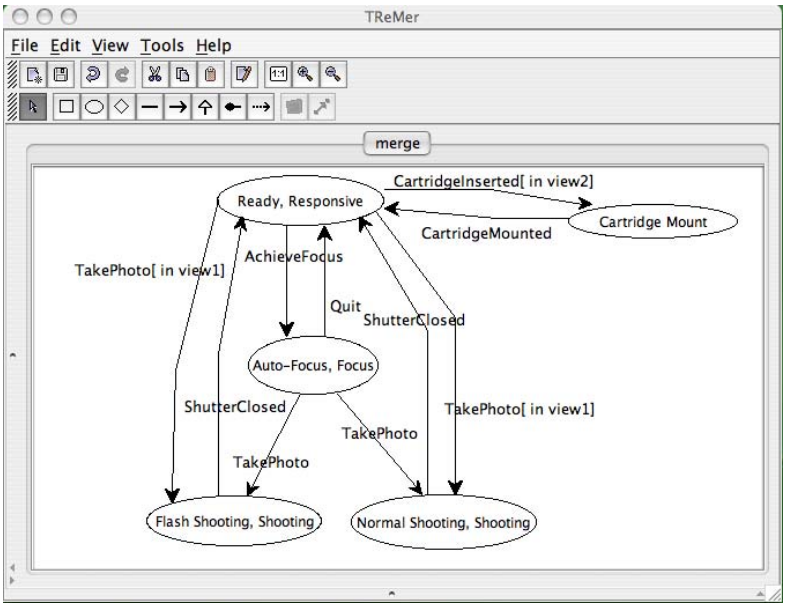


Figure 4.14: Behavioural merge of the models in Figure 4.11

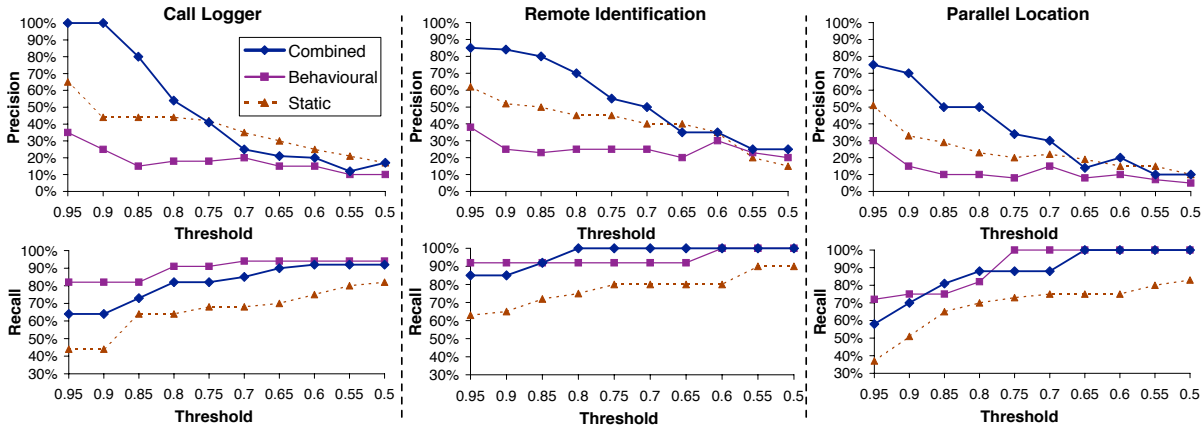


Figure 4.15: Results of static, behavioural, and combined matching.

hard to find by hand, for example where the models are complex, or the developers are less familiar with them. On the other hand, computing merge by hand is always likely to be laborious; our approach to merge is therefore useful if it produces semantically correct results and scales well.

Here, we present some initial steps to evaluate our work. First, we discuss the complexity of our Match and Merge operators, to show that they scale. We assess our Match

operator by measuring the accuracy of the relations it produces, in comparison with the assessment of a domain expert. We assess our Merge operator by proving that it preserves the behavioural properties of the input models.

4.8.1 Complexity

Let n_1 and n_2 be the number of states in the input models, and let m_1 and m_2 be the number of transitions in these models. The space and time complexities of computing typographic and linguistic similarity scores between individual pairs of name labels are negligible and bounded by a constant. The space complexity of Match is then the storage needed for keeping a state similarity matrix and a label similarity matrix (L in Section 4.5.2) and is $O(n_1 \times n_2 + m_1 \times m_2)$. The time complexity of static matching is $O(n_1 \times n_2)$ and of behavioural matching – $O(c \times m_1 \times m_2)$, where c is the maximum allowed number of iterations for the behavioural matching algorithm.

The space complexity of Merge is linear in the size of the correspondence relation ρ and the input models. Theoretically, the size of ρ is $O(n_1 \times n_2)$. In practice, we expect the size of ρ to be closer to $\max(n_1, n_2)$ giving us linear space complexity for practical purposes. This was indeed the case for our models (see Table 4.1). The time complexity of Merge is $O(m_1 \times m_2)$.

4.8.2 Accuracy of Match

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

We use two well-known metrics, namely, *precision*, and *recall*, to capture this intuition.

Precision measures quality (i.e., low number of false positives) and is the ratio of correct matches found to the total number of matches found. Recall measures coverage (i.e., low number of false 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 relation in Figure 4.8(b) and the desired relation is Figure 4.8(c), the precision and recall is 8/14 and 8/8, respectively.

A good matching technique should produce high precision and high recall. However, these two metrics tend to be inversely related: improvements in recall come at the cost of reducing precision and vice versa. The Software Engineering literature suggests that for information retrieval tasks, users are willing to tolerate a small decrease in precision if it can bring about a comparable increase in recall (Hayes *et al.*, 2003). We expect this to be true for model matching, especially for larger models: it is easier for users to remove incorrect matches rather than find missing ones. On the other hand, precision should not be too low. A precision less than 50% indicates that more than half of the found matches are wrong. In the worse case, it may take users more effort to remove incorrect matches and find missing correct matches than to do the matching manually.

We evaluated the precision and recall of our Match operator by applying it to a set of Statecharts models describing different telecom features at AT&T. We studied three pairs of models, describing variant specifications of telecom features at AT&T. One of these is the call logger feature in Section 4.1.1. Simplified versions of the variants of this feature were shown in Figure 4.1. 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 4.1, we show some characteristics of the studied models. For example, the first variant of the remote identification feature has 24 states and 44 transitions, and the second one has 19 states and 31 transitions. The correct relation (as identified manually

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

Table 4.1: Characteristics of the studied models.

by our domain expert) consists of 12 pairs of states. The Statecharts models of these features are available in Appendix A.

To compare the overall effectiveness of static matching, behavioural matching, and their combination, we compute their precision and recall for thresholds [3] ranging from 0.95 down to 0.5. The results are shown in Figure 4.15.

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.

Table 4.2 shows the precision-recall tradeoff points, which we believe to be reasonable for the studied examples. As shown in the table, for thresholds between 0.75 and 0.85, our combined matcher achieved a precision of more than 50% and a recall of more than 80%.

[3]Recall that threshold is the cutoff value used for determining the correspondence relation from the similarity degrees (see Section 4.5.4).

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

Table 4.2: Tradeoff precisions and recalls.

4.8.3 Correctness of Merge

In this section, we prove that our merge procedure described in Section 4.6.2 is behaviour-preserving. The proof goes by first flattening Statecharts models. In Definition 4.3.3, we provided a technique for converting Statecharts to flat state machines, i.e., LTSs. The procedure for flattening parameterized Statecharts is exactly the same, except that the result is a Mixed LTS rather than an LTS. Specifically, transitions labelled by a condition on the reserved variable **ID** are replaced by non-shared transitions, and the rest of the transitions – by shared ones. For example, Figure 4.6 shows the Mixed LTS corresponding to the parameterized model in Figure 4.9. It can be seen that guarded transitions in Figure 4.9 are replaced by non-shared transitions in Figure 4.6.

Given input Statecharts 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 4.8.1 *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 actions of L_1 and L_2, respectively. Then, $L_1 \preceq L_{1+2}@\{E_1 \uplus E_2 \setminus Act_2\}$ and $L_2 \preceq L_{1+2}@\{E_1 \uplus E_2 \setminus Act_1\}$.*

Before we give the proof, we provide an inductive definition, equivalent to Definition 4.3.5, for the refinement relation $\preceq$.

Definition 4.8.1 *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*

$$\forall s' \in S_1 \cdot \forall e \in E_1 \cup \{\tau\} \cdot s \xrightarrow{e}{}^{all} s' \Rightarrow \exists t' \in S_2 \cdot t \stackrel{e}{\Longrightarrow}{}^{all} t' \wedge s' \preceq^n t'$$

$$\forall t' \in S_2 \cdot \forall e \in E_2 \cup \{\tau\} \cdot t \xrightarrow{e}{}^{shared} t' \Rightarrow \exists s' \in S_1 \cdot s \stackrel{e}{\Longrightarrow}{}^{shared} s' \wedge s' \preceq^n t'$$

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}.

Note that in this proof, we assume that any typle $(s,t) \in \rho$ 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 4.8(c), which is defined between Statecharts in Figure 4.1, 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 4.5.

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 4.8.1, we need to show for every $(s,r) \in \rho_1$,

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

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

Here is the proof,

1. We identify four different 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 4.6.2 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 4.6.2)
$r = (s,t) \wedge \nexists t' \in S_2 \cdot (s',t') \in \rho \wedge (s,t) \xrightarrow{e}{}^{all} s'$

$\Rightarrow$ (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 4.6.2)
$r = s \wedge \exists t' \in S_2 \cdot (s',t') \in \rho \wedge s \xrightarrow{e}{}^{all} (s',t')$

$\Rightarrow$ (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 4.6.2)

$r = s \wedge \nexists t' \in S_2 \cdot (s',t') \in \rho \wedge s \xrightarrow{e}{}^{all} s'$

$\Rightarrow$ (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 $r' = s'$)

$\exists r' \in S_+ \cdot r \xrightarrow{e}{}^{all} r' \wedge s' \preceq^i r'$

2. By construction of merge in Section 4.6.2, 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 4.6.2)

$\exists (s,t),(s',t') \in \rho \cdot r = (s,t) \wedge r' = (s',t') \wedge s \xrightarrow{e}{}^{shared} s' \wedge e \neq \tau \bigvee$

$\exists (s,t),(s,t') \in \rho \cdot r = (s,t) \wedge r' = (s,t') \wedge s \xRightarrow{e}{}^{shared} s \wedge e = \tau$

$\Rightarrow$ (by definition of ρ and ρ_1)

$\exists s' \in S_1 \cdot s \xrightarrow{e}{}^{shared} s' \wedge (s',(s',t')) \in \rho_1 \wedge e \neq \tau \bigvee$

$s \xRightarrow{\tau}{}^{shared} s \wedge (s,(s,t')) \in \rho_1$

$\Rightarrow$ (by the inductive hypothesis)

$\exists s' \in S_1 \cdot s \xRightarrow{e}{}^{shared} s' \wedge s' \preceq^i r'$

The above proves that $\rho_1 \subseteq \bigcap_{i \geq 0} \preceq^i$. Since ρ_1 also relates s_0, i.e., the initial state of L_1, to (s_0,t_0), i.e., the initial state of L_{1+2}, ρ_1 is indeed a refinement relation between L_1 and L_{1+2}. Note that 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 above proof. □

Recall that by our definition in Section 4.6.2, 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 4.4, the input Statecharts 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 shared transitions

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

in Figure 4.1. 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 4.9, but the action callee=subscriber does not exist in Figure 4.1(b). Thus, we need to hide this action when comparing the merge with the model in Figure 4.1(b). Figure 4.16 shows the Mixed LTS corresponding to the merge in Figure 4.9 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 4.1(b) where the refinement relation is $\{((s,t),t) \mid (s,t)$ and t are states in Figure 4.9 and 4.1(b), respectively. $\}$.

By Theorems 4.3.1 and 4.8.1, 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

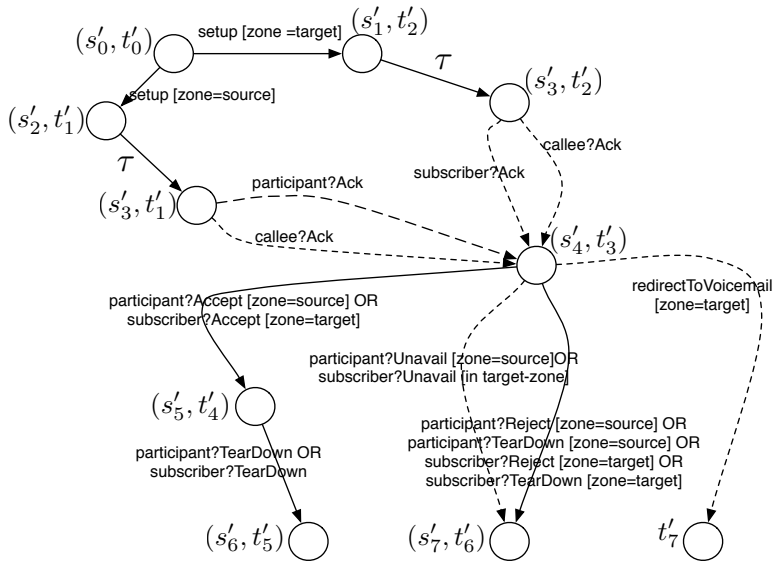


Figure 4.16: Mixed LTS which is equivalent to the Statecharts in Figure 4.9 except that actions callee=participant and callee==subscriber are hidden.

the behaviours of the individual input models. Therefore, any behaviour which is not present in neither of the input models is not present in 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.

(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. Thus, the merge preserves all positive traces of the input models.

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

As noted in Section 4.3, our models have deterministic semantics, achieved by assigning priority labels to transitions. Our merge construction respects transition priorities and ensures that merges are deterministic as well.

Section 4.6 described our procedure for merging *pairs* of models. This can be extended to n-ary merges by iteratively merging a new input model with the result of a previous merge, with two minor modifications: the reserved variable **ID** (in the merge procedure of Section 4.6.2) will range over subsets of the input model indices. In this case, the order in which the binary merges are applied does not affect the final result.

4.9 Related Work

In this section, we compare our work with the research on model matching and model merging in the areas of software engineering and databases.

4.9.1 Matching

Approaches to this problem can be categorized into two groups: *exact* and *approximate*. Exact matching is concerned with finding structural or behavioural conformance relations between models. Examples of structural relations are graph homomorphisms, and those of behavioural relations are bisimulation or simulation relations, or variants of these relations defined in process algebra (van Glabbeek, 1993). 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.

In any situation where models are developed independently, model matchers provide a way to discover the relationships between models, for example, to compare variants (Man-

delin *et al.*, 2006), to identify inconsistencies (Spanoudakis & Finkelstein, 1997), and to support reuse (Maiden & Sutcliffe, 1992). Sophisticated matching tools, e.g. (Bernstein *et al.*, 2004), can handle models that use different vocabularies and different levels of abstraction. In most of these applications, heuristic techniques are used 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 (Rahm & Bernstein, 2001). State-of-the-art schema matchers, such as Protoplasm (Bernstein *et al.*, 2004), combine several heuristics for computing similarities between schema elements. Our typographic and linguistic heuristics (Section 4.5.1) are very similar to those used in schema matching, but our other heuristics are tailored to behavioural models.

Several approaches to matching have been proposed in software engineering. (Maiden & Sutcliffe, 1992) employs heuristic reasoning for finding analogies between a problem description and already existing domain abstractions. (Ryan & Mathews, 1993) uses approximate graph matching for finding overlaps between concept graphs. (Alspaugh *et al.*, 1999) proposes term matching based on project glossaries for finding similarities between textual scenarios. (Mandelin *et al.*, 2006) combines diagrammatic and syntactic heuristics for finding matches between architecture models. None of these were specifically designed for behavioural models and are either inapplicable or unsuitable for matching Statecharts models.

In concurrency theory, several notions of behavioural conformance have been proposed to capture the behavioural similarity between models with quantitative features such as time or probability (van Breugel, 2008). 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., (de Alfaro *et al.*, 2004a; Sokolsky *et al.*, 2006)). Our formulation of behavioural similarity (Section 4.5.2) is analogous to these similarity relations. Their goal is to define a distance metric over the space of (quantitative) reactive processes and study the mathematical properties of the metric. Our goal, however, is to obtain a similarity measure that can detect pairs of states with a high degree of behavioural similarity.

4.9.2 Merging

Model merging spans several application areas. In database design, merge is an important step for producing a schema capturing the data requirements of all stakeholders (Bernstein, 2003). Software engineering deals extensively with model merging – several papers study the subject in specific domains, including early requirements (Sabetzadeh & Easterbrook, 2006), static UML diagrams (Alanen & Porres, 2003; Mehra *et al.*, 2005; Letkeman, 2006; Zito *et al.*, 2006), declarative specifications (Jackson, 2002), scenarios (Whittle & Schumann, 2000), and state-machines (Sabetzadeh & Easterbrook, 2003; Uchitel & Chechik, 2004).

Generally speaking, the above approaches can be categorized into two groups based on the mathematical machinery that they use to define and automate the merge process: (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 (Sabetzadeh & Easterbrook, 2006). 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

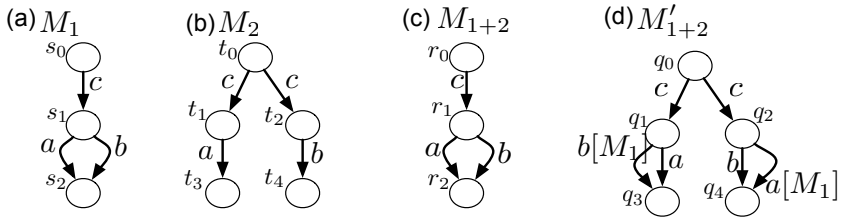


Figure 4.17: An example showing the difference between preservation of structure and behaviour: (a) model M_1; (b) model M_2; (c) a possible merge of M_1 and M_2 that preserves their behaviours; and (d) a possible merge of M_1 and M_2 that preserves their structure.

using a simple example. Consider the models M_1 and M_2 in Figures 4.17(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 4.17(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 4.17(d) is an alternative merge of M_1 and M_2 which is constructed based on the notion of refinement: It can be shown that M'_{1+2} refines both M_1 and M_2 with respect to the refinement given in Definition 4.3.5. 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 the notion of merge on refinement, we can choose to keep states distinct in the merged model even if ρ maps them to one single state in the other model. This is because refinement is more flexible than graph homomorphism. Homomorphisms do not allow us to map one state to several states because this deos not respect the structure of the underlying state machine graphs. Refinements, however, allow us to change the structure of the input models, e.g., we can duplicate states, as long as the behaviours of the models are properly lifted to their merge.

Approaches in the second group construct a merge as a common behavioural refinement of the original models. Larsen et. al. (Larsen *et al.*, 1995) show that this merge can

be logically characterized as conjunction of the original specifications, when models are consistent and have the same vocabulary. (Uchitel & Chechik, 2004) extends the work of (Larsen *et al.*, 1995) by providing support for merging models with different vocabularies. This is achieved by first hiding non-shared vocabulary using τ-actions, and then using a weaker notion of refinement, i.e., observational refinement, to relate models. This work supports incompleteness and can also detect inconsistencies; however, the merge operation fails when the models are inconsistent. Moreover, (Uchitel & Chechik, 2004) does not make model relationships explicit and matching is an implicit step in the merge process. This can make it difficult for modellers to guide the merge process as they cannot directly hypothesize the merge alternatives. (Fischbein & Uchitel, 2008) improves the work of (Uchitel & Chechik, 2004) partly by providing an algorithm for computing the *least* common (strong) refinement of partial models. However, like (Uchitel & Chechik, 2004), (Fischbein & Uchitel, 2008) does not handle inconsistent models. Finally, Huth and Pradhan (Huth & Pradhan, 2001) merge partial behavioural specifications where a dominance ordering over models is given to resolve some potential inconsistencies between models. These earlier approaches (1) do not provide support for merging models with variabilities (or behavioural inconsistencies), (2) neither handle hierarchical notations nor the question of reconciling the structural and behavioural aspects of state-machine merging, and (3) do not make model relationships explicit.

In this chapter, 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 model-based synthesis from scenarios (Uchitel & Chechik, 2004), program integration (Horwitz *et al.*, 1989), and merging declarative specifications (Jackson, 2002). While these approaches deal with different notations in settings different from ours, like our work their main challenge is preservation of semantics and support for handling inconsistencies.

Several approaches to variability modelling have been proposed in software mainte-

nance and product line engineering. For example, (Halmans & Pohl, 2003) 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 Statecharts models (Gomaa, 2004). A similar parameterization technique has been used in (Faulk, 2001) for capturing variability in SCR tables.

In requirements and early design model merging, variabilities are often treated as inconsistencies (Easterbrook & Chechik, 2001; Uchitel & Chechik, 2004; Sabetzadeh & Easterbrook, 2006). Some of these approaches require that only consistent models be merged (Uchitel & Chechik, 2004). However, others tolerate inconsistency, and can represent the inconsistencies explicitly in the resulting merged model (Easterbrook & Chechik, 2001; Sabetzadeh & Easterbrook, 2006). Our work is similar to the latter group as we explicitly model inconsistencies between models using parameterization.

4.10 Limitations

Our work has a number of limitations which we have listed below.

Our evaluation in Section 4.8 may not be a comprehensive assessment of the effectiveness of our Match operator: Firstly, in our evaluation, we assume that a matching relation agreeable to all users can be found. In practice, this may not be the case (Melnik, 2004). 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 4.8, 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 rela-

tions 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 the 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 whether this approach is feasible.

As noted in Section 4.6.2, shared parallel states are replaced with their semantically equivalent non-parallel structures. This may result in discontinuities between the conceptual structuring of the merge and that of the input models when parallel states have many substates. In our telecom models, parallel states have no more than a few substates each (less than five); therefore, the merged models still retained the essential structure of the input models. An alternative approach to handling parallel states may be needed for domains that make more extensive use of parallelization.

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

4.11 Conclusion

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

parameterized using guards on their transitions. The result is a merge that preserves the behaviours of the input models. We have developed a proof-of-concept implementation of these operators.

Models which have been developed distributedly may relate to one another in a variety of ways. The work presented in this chapter focuses on merging a collection of inter-related models when relationships describe overlaps between models' behaviours. Sometimes relationships describe behavioural interactions, in particular, when models are independent components or features of a system. In situations where models are interacting, Match describes how models interact through their interfaces, (e.g., how messages are communicated, or how data is exchanged between models), and Merge provides a way to combine models with respect to the semantics of communication. A pervasive problem in this context is that, while individual models behave correctly, their interaction may result in undesired behaviours. In Chapter 5, we study this problem and propose a technique for constructing a system of interacting models such that their undesirable behaviours are not retained.

Chapter 5

Composing Features and Analysing Interactions

5.1 Introduction

Synthesis of system compositions from a given set of features is an important and very challenging problem. In this chapter, we make a step towards this goal by describing an efficient technique for synthesizing feature-based systems that are arranged in a pipeline architecture. We identify and formalize a design pattern that is commonly used in feature-based development. We show that this pattern enables *compositional* synthesis of feature arrangements. In particular, the pattern allows us to add or remove features from an existing system without having to reconfigure the system from scratch. We describe an implementation of our technique and evaluate its applicability and effectiveness using a set of telecommunication features from AT&T, arranged within the DFC architecture.

5.1.1 Synthesizing Feature-based Systems

Feature-based development has long been used as a way to provide separation of concerns, to facilitate maintenance and reuse, and to support software customization based on

end-user needs (Prehofer, 1997; Li *et al.*, 2002; Harrison *et al.*, 2002; Batory, 2004; Li *et al.*, 2005). Individual features typically capture specific units of functionality with direct relationships to the requirements that inspired the software system in the first place (Fisler & Krishnamurthi, 2005). By closely mirroring requirements, features make it easier to reconfigure or expand a system as its underlying requirements change over time.

To meet the desirable properties expected from a feature-based system, the interactions among its features need to be constrained and orchestrated. This is often done by putting features in a suitable arrangement, typically a linear one such as a stack or a pipeline, that inhibits undesirable interactions.

Existing research on feature interaction analysis, e.g., (Pomakis & Atlee, 1996; Jackson & Zave, 1998; Hay & Atlee, 2000; Hall, 2000; Plath & Ryan, 2001; Li *et al.*, 2002; Felty & Namjoshi, 2003; Bruns, 2005), largely concentrates on reasoning about and resolving undesirable interactions between a set of features *whose arrangement is given a priori*. Yet a complementary problem, of how to automatically synthesize an arrangement when one is not given, has not been studied much. The problem is important – it currently takes substantial expertise and effort to find an arrangement of features that does not result in undesirable interactions.

Unfortunately, a naive attempt at automatically arranging features is infeasible: there is an exponential number of alternative arrangements to consider when searching for a desirable one. Hence, we need compositional techniques that can reduce the problem of finding a desirable arrangement into smaller subproblems. This need becomes even more pressing in systems that evolve over time, where features are periodically added, removed, or revised. Without compositional techniques for synthesizing evolving systems, new arrangements of features may have to be created from scratch after each change.

Our goal is to provide compositional techniques for synthesizing software systems from an *evolving*, *arbitrarily large* set of (*different*) features. To achieve this goal, we

draw inspiration from the literature on component-based software. A general way to enable compositional reasoning about systems with an arbitrary number of components is by exploiting behavioural similarities between components. For example, (Emerson & Namjoshi, 2003; Emerson & Kahlon, 2000) show that system-wide verification tasks can be decomposed if components exhibit identical or virtually identical behaviours. The motivation for the work is verification of low-level operating system protocols, e.g., mutual exclusion where several identical copies of a process attempt to enter a critical section. More recent work, e.g., (Betin-Can *et al.*, 2005), explores similar ideas to bring compositional reasoning to software systems in which components have diverse behaviours. There, the required degree of similarity between components is achieved by having components implement a *design pattern*.

5.1.2 Contributions of This Chapter

In this chapter, we aim to study how the design patterns used in feature-based development can enable compositional synthesis of feature arrangements. We ground our work on *pipelines* – popular architectures for building feature-based systems (Braithwaite & Atlee, 1994; Jackson & Zave, 1998; Hay & Atlee, 2000; Li *et al.*, 2002) which allow one to define the overall behaviour of a system in terms of a simple composition of the behaviours of the individual features (Shaw & Garlan, 1996).

A common objective in designing feature pipelines is to minimize the visibility of each feature to the rest. This is to ensure that individual features can operate without relying on those appearing before or after them in the pipeline (Shaw & Garlan, 1996). To realize this objective, features are usually designed so that they engage in defining the overall behaviour of the system only when their function is needed. More precisely, features alter the flow of signals in the pipeline only when they are providing their service; otherwise, they let the signals pass through without side-effects. The ability of a feature to remain unobservable to other features when it is not providing its service is called *transparency*.

We argue that transparency is sufficient to make pipeline synthesis compositional, requiring the analysis of just pairs of features to determine their relative order in the overall pipeline. In particular, we make the following contributions:

1. We formalize the transparency pattern of behaviour and show that for features implementing this pattern, *global* constraints can be inferred on the order of the features through *pairwise* analysis of the features.

2. We describe a *sound* and *complete* compositional algorithm for synthesizing pipeline arrangements. Given a set of features and a set of safety properties describing undesirable interactions, our algorithm computes an arrangement of the features that is safe for the given properties. Specifically, the algorithm uses the safety properties to compute a set of pairwise ordering constraints between the features. Due to the transparent behaviour of the features, any global ordering that violates a pairwise ordering constraint can be deemed unsafe and pruned from the search space of the solution, leaving a relatively small number of global orderings to be generated and verified by the algorithm. Our algorithm is *change-aware* in the sense that after adding or modifying a feature, we need to update only the pairwise ordering constraints related to that particular feature and reuse the remaining constraints from the previous system.

3. We report on a prototype implementation of our synthesis algorithm, applying it to a set of AT&T telecom features to find a safe arrangement for them in the Distributed Feature Composition (DFC) architecture (Jackson & Zave, 1998). Our algorithm could automatically and efficiently compute a safe arrangement for the DFC features in our study.

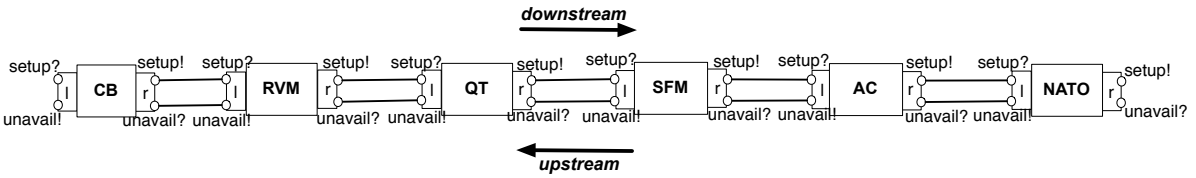


Figure 5.1: A simplified linear DFC scenario.

5.1.3 Organization of This Chapter

The rest of the chapter is organized as follows. In Section 5.2, we motivate our work using an example from the telecom domain. In Section 5.3, we formalize features as I/O automata and define a notion of binding for describing pipelines. Section 5.4 is the main contribution of this chapter. It formalizes the transparency pattern that guarantees that synthesis can be done compositionally. We describe our synthesis algorithm in Section 5.5 and its implementation in Section 5.6. In Section 5.7, we evaluate our technique on a set of AT&T telecom features. We review related work and compare it with our approach in Section 5.8. We discuss the limitations of our approach in Section 5.9, and conclude this chapter with a summary of contributions and a discussion on potential extensions of this work in Section 5.10.

5.2 Motivation

We motivate our work by analyzing a simplified instance of a telecom scenario (see Figure 5.1). Features in this scenario are arranged in a pipeline and include Call Blocking (CB), Record Voice Mail (RVM), Quiet Time (QT), Sequential Find Me (SFM), No Answer Time Out (NATO), and Answer Confirm (AC).

Pipeline features communicate by passing signals to their immediate neighbours. Signals that travel end-to-end pass through all features, allowing each feature to perceive

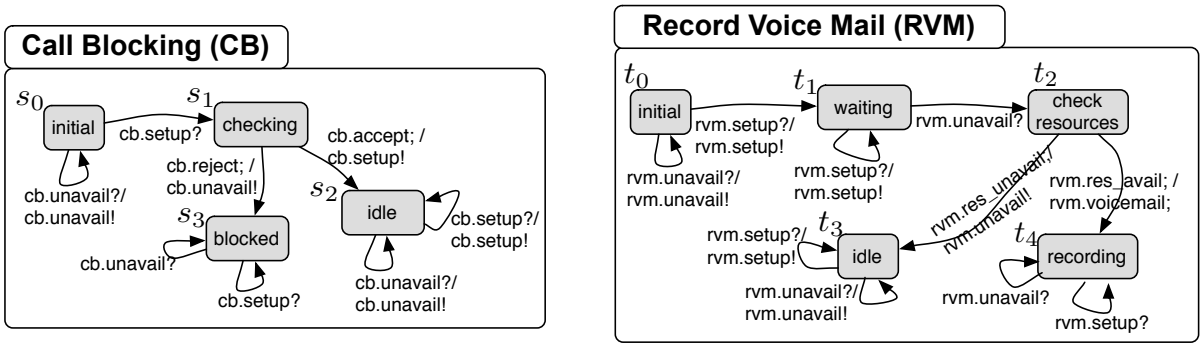


Figure 5.2: Call Blocking (CB) and Record Voice Mail (RVM).

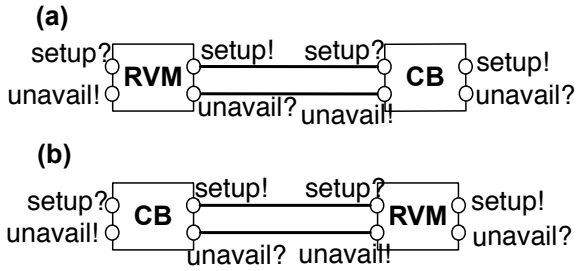


Figure 5.3: Possible orderings of the features in Figure 5.2.

and modify the overall function of the pipeline. For example, Figure 5.1 shows the flow of the signals setup and unavail. There are many other signal types, but we show only the most relevant ones here.

The communication between the features in a pipeline is either buffered or unbuffered (synchronous). The former facilitates reliable communication but complicates reasoning: it is known that verification of a distributed system with unbounded buffers is undecidable (Brand & Zafiropulo, 1983). Instead, we assume that features communicate through synchronous message passing, which makes for more tractable reasoning but imposes restrictions on the design of features: they should be responsive to *all* potential input *at all time*, i.e., they should be *input-enabled*. This requirement is captured in a number of standard formalisms for describing concurrent systems, e.g., I/O automata (Lynch

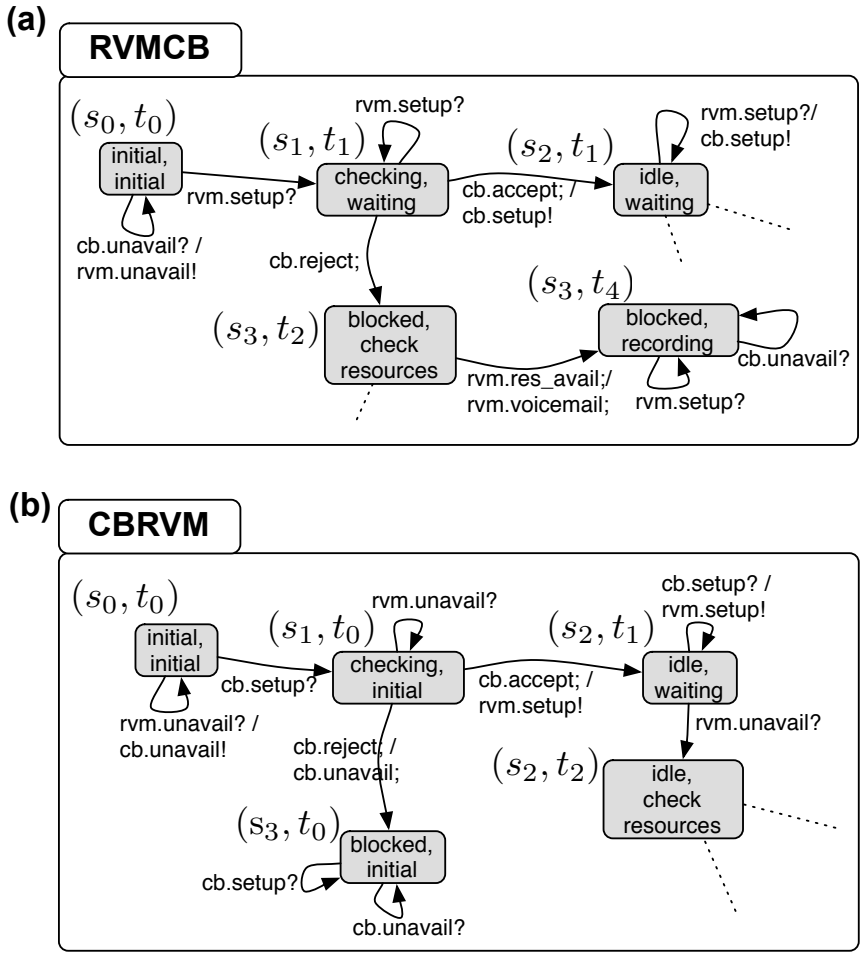


Figure 5.4: Fragments of the compositions of the features in Figure 5.2 with respect to the orderings in Figure 5.3.

& Tuttle, 1987). For example, all the features in Figure 5.1 are enabled for setup and unavail.

To refer to the directions within a pipeline, we use the terms *upstream* (right to left) and *downstream* (left to right). In our example, the setup signal travels downstream, and the unavail signal travels upstream. For features F and F' in a pipeline, we write $F < F'$ to indicate that F is upstream ("to the left of") of F'. For example, in Figure 5.1, CB $<$ NATO.

Figure 5.2 shows the state machines for CB and RVM in the pipeline of Figure 5.1. The purpose of CB is to block calling requests coming from addresses on a blocked list. CB becomes active by receiving a setup signal containing initialization data such as the directory numbers of the caller and callee. Using this data and its internal logic, CB decides whether the caller should be blocked. If so, it moves to the blocked state and tears down the call; otherwise, it moves to the idle state and effectively becomes invisible. The purpose of RVM is to record a voicemail message when the callee is not available. Like CB, RVM is activated on receipt of a setup signal. It then remains in its waiting state until it receives an unavail signal, indicating that the callee is unavailable or is unable to receive the call. If the media resource is available, RVM moves to the recording state and lets the caller leave a voicemail message. Otherwise, if the media resource is unavailable, e.g., the mailbox quota for the user is exceeded, RVM moves to its idle state.

In Figure 5.2, a label "e1/e2" on a transition indicates that the transition is triggered by action "e1" and generates action "e2" after being taken. Transitions can be triggered either by input actions, those received from the outside, or by internal actions. When taken, a transition generates zero or more internal or output actions. It is assumed that the actions generated by a state machine do not trigger any transition of that state machine. To distinguish between input, output, and internal actions, we append to each action e the symbol "?" if e is an input action, the symbol "!" if e is an output action, and the symbol ";" if e is an internal action. Further, to disambiguate between the actions of different state machines, we prefix every action with the name of the state machine it belongs to.

Feature Interaction. The behaviour of the composition of the features in a pipeline depends on the ordering of the features, and the goal of our work is to synthesize an ordering which will guarantee absence of undesirable compositions. For example, suppose we are trying to avoid the composition: "RVM should not record a message if CB blocks

the caller" (Zave, 1999), formalized as the following negative trace[1]:

$$NS_1 = \mathsf{cb.reject}; \mathsf{rvm.voicemail};$$

CB and RVM can be put in a pipeline one of the two ways, shown in Figure 5.3. The ordering in Figure 5.3(a) yields the composition in Figure 5.4(a), and the one in Figure 5.3(b) – the composition in Figure 5.4(b). These compositions were computed based on the parallel composition semantics in synchronous mode of communication (Milner, 1989). The composition in Figure 5.4(a) results in an undesirable interaction: the path from (s_0, t_0) to (s_3, t_4) generates the trace NS_1, i.e., "rvm.voicemail;" comes after "cb.reject;". The composition in Figure 5.4(b), on the other hand, does not exhibit NS_1, implying that CB should come before RVM in a pipeline. Note that since the size of these compositions is huge, Figures 5.4(a) and (b) only show the relevant fragments of these compositions.

Synthesis Challenge. In general, finding a suitable ordering cannot be done compositionally when the features in a pipeline have unconstrained designs. For example, consider sample features A and B in Figure 5.5(a) and the property "A voicemail message should not be recorded"[2], i.e., action "b.voicemail;" must not be produced by the composition of A and B. This property does not hold over either a pipeline in which $A < B$, or the one in which $B < A$. In the former case, A sends setup to B, and B generates "b.voicemail;", and in the latter case, B receives setup from the environment and generates "b.voicemail;". So, it may seem that the given correctness property does not hold on a pipeline containing A and B. However, consider the new feature C in Figure 5.5(b) which blocks the action setup. The pipeline $A < C < B$ in Figure 5.5(b) satisfies the given correctness property, i.e., the composition of these features, when arranged in the above order, does not generate "b.voicemail;". This example shows that,

[1]The trace "rvm.voicemail; cb.reject;" could have been considered instead of NS_1 as well.
[2]This property is used only for illustration.

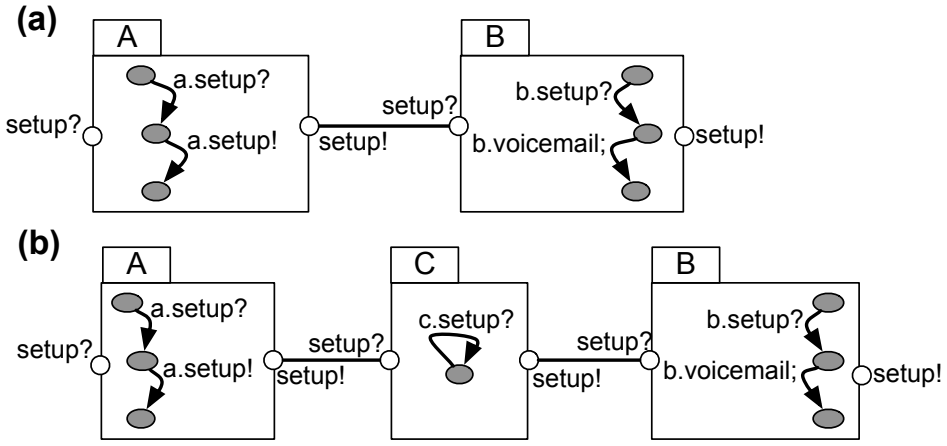


Figure 5.5: Local ordering vs. global ordering.

in general, we may not be able to infer a global ordering over the pipeline by analyzing subsets of components. Even though the given correctness property only concerns B, our analysis needs to consider *all* the components in the pipeline. Hence, given n unrestricted components, we need to check exponentially many ($n! \approx O(2^{n \log n})$) pipeline arrangements to find one which satisfies the properties of interest. This is intractable for all but the most trivial pipelines.

Transparency Pattern. To be able to lift an ordering over a subset of pipeline features to the entire pipeline, we rely on a pattern of behaviour called *transparency*. Each feature implementing this pattern can exhibit an execution along which it is unobservable (transparent). When executing transparently, a feature sends any signal received from its left to its right, and any signal received from its right to its left, possibly with some finite delay. Features implementing the transparency pattern can still perform their specific functionality via other executions, or via unobservable behaviours.

For example, in Figure 5.2, CB's transparent execution is from s_0 to s_2, and RVM's – from t_0 to t_3. CB behaves transparently if the call request comes from a non-blocked address. In this case, the system proceeds as if CB were never present; otherwise, CB

provides its service by blocking the incoming call, i.e., by taking the path from s_0 to s_3. As for RVM, the feature exhibits its transparent behaviour when its media resource is unavailable; otherwise, it allows the user to leave a voicemail message by taking the path from t_0 to t_4.

For pipeline features implementing the transparency pattern, we prove the following (Section 5.4): if a pipeline consisting of just two features F and F' where $F < F'$ violates a safety property φ, a pipeline with an arbitrary number of components in which $F < F'$ violates φ as well. This enables a compositional algorithm for synthesizing pipeline orderings (Section 5.5).

5.3 I/O Automata and Pipelines

We describe features as I/O automata (Lynch & Tuttle, 1987). This formalism is chosen because

1. I/O automata allow distinguishing between the input, internal, and output actions of features – this distinction between different types of actions is crucial for properly describing the communications between features (Lynch & Tuttle, 1987); and

2. I/O automata are input-enabled by design. Input-enabledness makes it easier to detect and avoid deadlocks (Lynch & Tuttle, 1987; Zave & Jackson, 2002) and further, provides a way to terminate features that are stuck in error loops and hence are wasting resources (Zave & Jackson, 2002).

Definition 5.3.1 (I/O automata (Lynch & Tuttle, 1987)) *An* I/O automaton *is a tuple $A = (S, s_0, E, R)$, where S is a finite set of states; $s_0 \in S$ is an initial state; E is a set of actions partitioned into input actions (E^i), output actions (E^o), and internal actions (E^h); and $R \subseteq S \times E \times S$ is a set of transitions.*

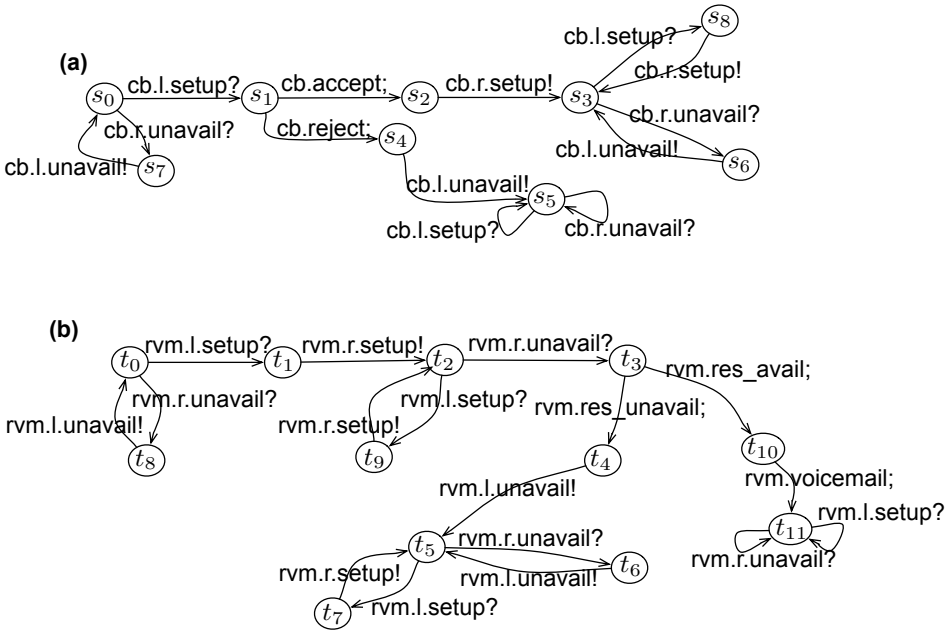


Figure 5.6: I/O automata for the state machines in Figure 5.2.

Input actions are those that a feature receives from its environment. Internal actions represent events scoped inside a feature and invisible outside of it. Examples of such events include internal timers and communication with media devices. Output actions represent a feature's response to its input and internal actions.

An I/O automaton can be viewed as an LTS if the distinction between input, output and internal actions is ignored. Given an I/O automaton $A = (S, s_0, E = E^i \cup E^o \cup E^h, R)$, we write $LTS(A)$ to denote the LTS (S, s_0, E, R). Similar to LTSs, we write $A@E'$ to denote A with its set of actions reduced from E to E', and write $\mathcal{L}(A)$ to denote the set of traces of A. Figures 5.6(a) and (b) show the I/O automata for the state machines in Figures 5.2(a) and (b), respectively. The labels of the input and output actions of these I/O automata have infixes "r" (right) and "l" (left); these indicate the directions in which these actions are communicated (see Definition 5.3.2).

We say a state s is *enabled* for an action e if s has an outgoing transition labelled e. A state s is *quiescent* if s is not enabled for any output or internal actions. Intuitively, an automaton in a quiescent state is strictly waiting for an input from its environment. An I/O automaton A is *input-enabled* if the following conditions hold:

1. A returns promptly to some quiescent state after leaving one. We assume the execution time of transitions labelled with output and internal actions to be negligible. Thus, prompt return to a quiescent state means that output and internal actions never block the execution, and further, no cycle of transitions labelled with only internal and output actions exists.

2. Quiescent states of A are enabled for all input actions. For example, states s_0, s_3, and s_5 in Figure 5.6(a) are quiescent and are enabled for all input actions of CB, i.e., "cb.l.setup?" and "cb.r.unavail?".

As shown in Figure 5.1, each feature has one port on its left and one on its right side, and actions can be sent or received from either of these two ports. To be able to refer to the direction of communication in a pipeline, we augment I/O automata with action mappings which specify the port from which an action is sent or received.

Definition 5.3.2 (Features) *A* feature F *is a tuple* (A_F, f) *where* A_F *is an I/O automaton, and* $f : E^i \cup E^o \to \{\mathsf{r}, \mathsf{l}\}$ *is a function that maps every input and output action of* F *to either the right,* r*, or the left,* l*, port of* F*. We write "*F.r.e*" (or, respectively, "*F.l.e*") to say that action* e *is mapped to port "*r*" (or, respectively, "*l*").*

Note that f does not map the internal actions of a feature because these actions are invisible outside the feature.

In Figure 5.1, the smaller boxes attached to the features denote the ports. Actions are visualized as small circles on the appropriate ports. For example, CB has an (output) action "cb.r.setup!" mapped to its right port, and RVM has an (input) action "rvm.l.setup?" mapped to its left port.

To formally specify how two consecutive features in a pipeline communicate, we define a notion of *binding* for connecting the right port of one feature to the left port of another.

Definition 5.3.3 (Pipeline Bindings) *Let F_1 and F_2 be consecutive features in a pipeline; let $R = \{e \in E_1 \mid f_1(e) = \mathsf{r}\}$; and let $L = \{e \in E_2 \mid f_2(e) = \mathsf{l}\}$. A* (pipeline) binding $B \subseteq R \times L$ *between F_1 and F_2 is a one-to-one correspondence relation between L and R that relates input actions only to output actions, and output actions only to input actions; i.e.,*

$$(e_1, e_2) \in B \implies \big((e_1 \in E_1^o \wedge e_2 \in E_2^i) \vee (e_1 \in E_1^i \wedge e_2 \in E_2^o)\big)$$

For a binding B, we say an action is shared *if it occurs in some tuple of B, and* non-shared *otherwise.*

The links between the features in Figure 5.1 can be expressed as bindings. For example, the CB–RVM link in the figure is characterized by the following binding:

$$B = \{(\mathsf{cb.r.setup!}, \mathsf{rvm.l.setup?}), (\mathsf{cb.r.unavail?}, \mathsf{rvm.l.unavail!})\}$$

which indicates that the output action "cb.r.setup!" (of CB) synchronizes with the input action "rvm.l.setup?" (of RVM), and the input action "cb.r.unavail?" (of CB) synchronizes with the output action "rvm.l.unavail!" (of RVM).

In our working example, bindings are meaningful only if they relate actions with identical signal names. For example, had we considered an additional upstream-traveling signal unknown in Figure 5.1, it would have been incorrect to, say, relate actions "cb.r.unknown?" and "rvm.l.unavail!". Thus, in this chapter we assume that features use a unified set of signals and all bindings are based on signal name equivalences. On the other hand, we recognize that there may be domains where this assumption does not hold: features may refer to a shared signal by different names, or refer to non-shared signals by the same name. Through making mappings between actions explicit, all such bindings can be captured by Definition 5.3.3 directly.

To obtain the overall behaviour of a set of communicating features, we compose them with respect to the bindings established between them. To this end, we define a parallel composition of I/O automata, whereby features synchronize their shared actions and interleave their non-shared ones.

Definition 5.3.4 (Composition of Pipeline Features) *Let F_1 and F_2 be consecutive features in a pipeline linked by a binding B. The* parallel composition *of F_1 and F_2 with respect to B, denoted $F_1||_B F_2$, is a feature (A, f) where*

- $A = (S_1 \times S_2, (s_0, t_0), E = E^i \cup E^o \cup E^h, R)$ *with E^i, E^o, E^h, and R defined as follows:*

$$E^i = (E_1^i \cup E_2^i) \setminus \{e \mid e \text{ is a shared input action}\}$$

$$E^o = (E_1^o \cup E_2^o) \setminus \{e \mid e \text{ is a shared output action}\}$$

$$E^h = (E_1^h \cup E_2^h) \cup B$$

$$\begin{aligned} R = {} & \{((s,t), e, (s',t)) \mid (s, e, s') \in R_1 \wedge e \text{ is a non-shared action}\} \bigcup \\ & \{((s,t), e, (s,t')) \mid (t, e, t') \in R_2 \wedge e \text{ is a non-shared action}\} \bigcup \\ & \{((s,t),(e,e'),(s',t')) \mid (s,e,s') \in R_1 \wedge (t,e',t') \in R_2 \wedge (e,e') \in B\} \end{aligned}$$

- $f = (f_1 \cup f_2) \setminus \{e \mid e \text{ is a shared action}\}$

The above is the same as the standard definition of parallel composition for I/O automata (Lynch & Tuttle, 1987), except that we use bindings to explicitly specify the shared actions prior to composition. Since bindings are one-to-one, it easily follows that the $||_B$ operator is associative. Thus, the global composition of the features in a pipeline can be formulated as a series of binary compositions.

5.4 Formalizing Transparency

Intuitively, if a feature F implements the transparency pattern (motivated in Section 5.2), then there is *some* environment that coerces F to exhibit its transparent behaviour. For

example, CB (in Figure 5.2) exhibits its transparent execution, i.e., from s_0 to s_2, when data from the environment indicates that the callee has not blocked the caller. Since pipeline features act independently (Jackson & Zave, 1998; Shaw & Garlan, 1996), each feature can be coerced into its transparent execution independently of other features.

In this section, we formalize the above intuition and prove (in Theorem 5.4.1) that if all features implement the transparency pattern, the following holds:

"If a pipeline with two features (F followed by F') violates a safety property φ, a pipeline with an arbitrary number of features in which $F < F'$ violates φ as well."

We exploit this result in Section 5.5 to provide a compositional algorithm for ordering features in a pipeline.

The formalization of the transparency pattern G is shown in Figure 5.7. It is expressed as an I/O automaton with generic input actions $G.\mathsf{l}.\langle x\rangle?$ and $G.\mathsf{r}.\langle y\rangle?$, and generic output actions $G.\mathsf{l}.\langle y\rangle!$ and $G.\mathsf{r}.\langle x\rangle!$. State S_0 is quiescent, and states S_1 and S_2 are transient. A feature implementing this pattern can exhibit some execution along which it forwards any signal it receives from its left port onto its right port, and vice versa. On this execution, a feature can delay the transmission of actions for a *finite* amount of time to perform its internal behaviours, but is not allowed to add or omit any actions, or to change the order of actions being transmitted. If the environmental data is such that a feature has to provide its service in response, the feature chooses a non-transparent execution or simply fulfills its functionality through internal actions on its transparent execution.

The cycle between states S_0 and S_1 in Figure 5.7 (hereafter, the *downstream cycle*) handles signals that travel downstream, and the cycle between S_0 and S_2 (hereafter, the *upstream cycle*) handles signals traveling upstream. To adapt the generic transparency pattern to a specific pipeline problem, we need a copy of the downstream cycle for every signal traveling downstream, and a copy of the upstream cycle for every signal traveling upstream. For example, Figure 5.8 shows the adaptation, P, of the pattern to the pipeline in Figure 5.1. Since this pipeline has one downstream traveling signal, `setup`, and one

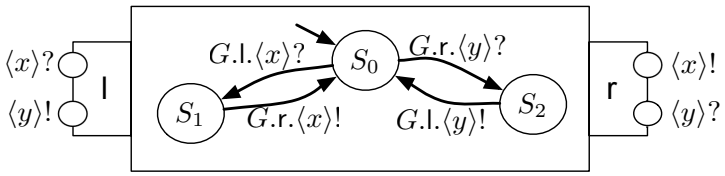


Figure 5.7: G: Generic transparency pattern.

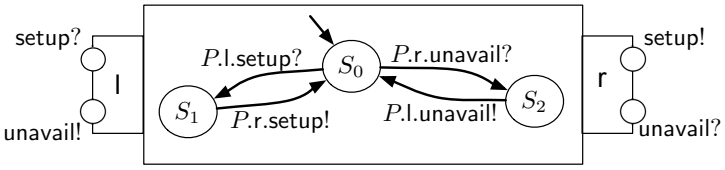


Figure 5.8: P: Adaptation of the generic transparency pattern to the pipeline in Figure 5.1.

upstream traveling signal, unavail, P has one copy of the downstream and one copy of the upstream cycle. Had we considered further signals, we would have had more copies of the corresponding cycles in this adaptation.

We characterize the implementation relation between a feature and its adaptation by weak simulation (see Definition 2.3.3), which allows us to relate features with different sets of actions. Having such flexibility is key: although the features in a pipeline share the same input and output actions with the pattern adaptation, each feature has its own set of internal actions. For example, consider features CB and RVM in Figure 5.6. CB's internal actions are "cb.reject;" and "cb.accept;", whereas RVM's are "rvm.res_unavail;", "rvm.res_avail;" and "rvm.voicemail;". Such internal actions are not used in P in Figure 5.8.

To establish a simulation relation between a feature and its pattern adaptation, we need to hide the feature's internal actions. For example, after replacing actions "cb.reject;" and "cb.accept;" of CB and "rvm.voicemail;" of RVM with τ, both CB and

RVM simulate P. The simulation relation for CB is

$$\{(s_0, S_0), (s_1, S_1), (s_2, S_1), (s_3, S_0), (s_6, S_2), (s_7, S_2), (s_8, S_1)\}$$

and for RVM is

$$\{(t_0, S_0), (t_1, S_1), (t_2, S_0), (t_3, S_2), (t_4, S_2), (t_5, S_0), (t_6, S_2),$$
$$(t_7, S_1), (t_8, S_2), (t_9, S_1)\}$$

Before giving the main result of this section, Theorem 5.4.1, we state two lemmas used in the proof of the theorem. For the remainder of this section, let P be the adaptation of the generic transparency pattern, G, for a particular pipeline.

Lemma 1 *Let F be a feature, and let B_1 bind F.r to P.l. If F violates a desired safety property, so does $F||_{B_1}P$. Similarly, let B_2 bind P.r to F.l. If F violates a desired safety property, so does $P||_{B_2}F$.*

Proof:

We provide the proof for the first case, i.e., binding B_1. The proof for the second case, i.e., binding B_2, is symmetric. The proof follows from the following two steps:

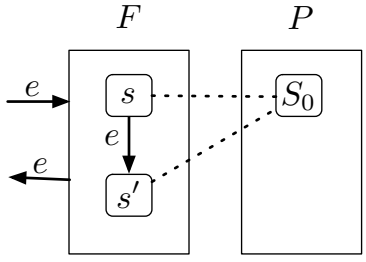
I. We first show that $F \preceq F||_{B_1}P$.

Let $F = (\Sigma_1, s_0, R_1, E_1)$, and let $P = (\Sigma_2 = \{S_0, S_1, S_2, \ldots, S_n\}, S_0, R_2, E_2)$ where S_0 is quiescent and $S_1, \ldots, S_n$ are non-quiescent (e.g., see Figure 5.8). Define $\Phi \subseteq (\Sigma_1 \times (\Sigma_1 \times \Sigma_2))$ as follows:

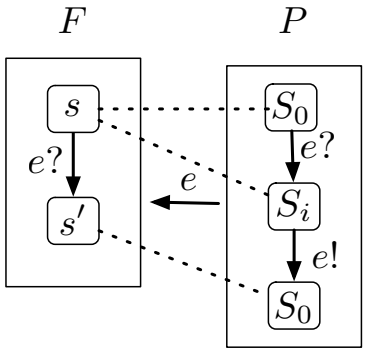
1. $(s_0, (s_0, S_0)) \in \Phi$ and

2. For every $s, s' \in \Sigma_1$ such that $s \xrightarrow{e} s'$,

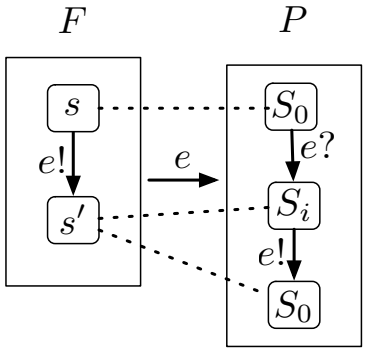
- **Internal:** if e is an internal action, or is an input action received from left, or is an output action sent to left; then (1) $(s, (s, S_0)) \in \Phi$; and (2) $(s', (s', S_0)) \in \Phi$.



-**Input:** if e is an input action received from right; then (1) $(s,(s,S_0)) \in \Phi$; (2) $(s,(s,S_i)) \in \Phi$ such that $S_i \in \Sigma_2$ and $S_0 \xrightarrow{e?} S_i$; and (3) $(s',(s',S_0)) \in \Phi$.



-**Output:** if $e!$ is an output action sent to right; then (1) $(s,(s,S_0)) \in \Phi$; (2) $(s',(s',S_i)) \in \Phi$ such that $S_i \in \Sigma_2$ and $S_0 \xrightarrow{e?} S_i$; and (3) $(s',(s',S_0)) \in \Phi$.



By Definition 5.3.4, $F||_{B_1}P = (\Sigma_1 \times \Sigma_2, (s_0,S_0), R_{||}, E_1 \cup E_2)$. Let Σ_3 be the set of states of $F||_{B_1}P$ reachable from (s_0,S_0). We first note that $\Phi \subseteq \Sigma_1 \times \Sigma_3$. This is because the above construction follows the definition of parallel composition (Definition 5.3.4). More specifically,

Internal: I/O automaton F moves on its action e; and P remains on its state S_0.

Input: I/O automaton P receives $e?$ from its right and moves to state S_i. Then, P moves from S_i back to S_0, sending action $e!$ to its left (i.e., to F). This causes F to move from s to s' on its input action $e?$.

Output: I/O automaton F moves on its output action $e!$, sending action e to P. This causes P to move from S_0 to S_i where $S_0 \xrightarrow{e?} S_i$. Then, P sends $e!$ to its right and moves back to S_0.

We argue that Φ is a simulation between F and $F||_{B_1}P$. By our construction of Φ, for every $s \in \Sigma_1$ reachable from s_0, there is a tuple $(s,(s,S_0)) \in \Phi$. It is easy to show that every state s of F is simulated by state (s,S_0) of $F||_{B_1}P$. Thus, s_0 is simulated by (s_0,S_0), and hence, Φ is a simulation relation between F and $F||_{B_1}P$.

II. Let σ_{neg} be a negative trace. By **I.**, we have $\mathcal{L}(F) \subseteq \mathcal{L}(F||_{B_1}P)$. Thus, $\sigma_{neg} \in \mathcal{L}(F)$, implies that $\sigma_{neg} \in \mathcal{L}(F||_{B_1}P)$. Therefore, if F violates a safety property, so does $F||_{B_1}P$.

□

The following lemma states that if the features in a pipeline implement the transparency pattern, so does the entire pipeline. That is, the features cannot prohibit one another from exhibiting their transparent behaviour.

Lemma 2 *Let $F_1, \ldots, F_n$ be consecutive features in a pipeline, where B_i binds F_i.r to F_{i+1}.l. If every F_i $(1 \leq i \leq n)$ implements (i.e., weak simulates) P, so does the composition*

$$F_1||_{B_1}F_2||_{B_2}\ldots||_{B_{n-1}}F_n.$$

Proof:

We first recall two standard results on parallel composition of state transition systems (see (Clarke *et al.*, 1999)).

(1) for every M_1, M_2 and M_3, if $M_1 \preceq M_2$ then $M_3||M_1 \preceq M_3||M_2$.

(2) for every M, we have $M \preceq M||M$

The proof follows by induction on n. The base case, $n = 1$, is trivial. Let $F = F_1||_{B_1}F_2||_{B_2}\ldots||_{B_{n-2}}F_{n-1}$.

$$\begin{aligned}
& P \preceq F_1 \wedge \ldots \wedge P \preceq F_n \\
& \text{(by the inductive hypothesis)} \\
\Rightarrow\ & P \preceq F \wedge P \preceq F_n \\
& \text{(by } \mathbf{(1)}\text{)} \\
\Rightarrow\ & P||_{B_{n-1}}F_n \preceq F||_{B_{n-1}}F_n \wedge P||_{B_{n-1}}P \preceq P||_{B_{n-1}}F_n \\
& \text{(by transitivity of } \preceq\text{)} \\
\Rightarrow\ & P||_{B_{n-1}}P \preceq F||_{B_{n-1}}F_n \\
& \text{(by } \mathbf{(2)}\text{)} \\
\Rightarrow\ & P \preceq F||_{B_{n-1}}F_n
\end{aligned}$$

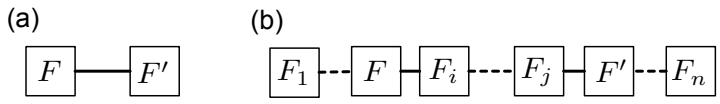


Figure 5.9: An illustration for Theorem 5.4.1.

Note that the actions of the left and right ports of all features $F_1, \ldots, F_n$ are the same as those of P. Thus, all bindings $B_1, \ldots, B_n$ are identical. Therefore, for any B_i, the operator $||_{B_i}$ can be used to compose any pair of features or any feature with P. □

Finally, we present the main theorem of this section:

Theorem 5.4.1 *Let $F, F', F_1, \ldots, F_n$ be pipeline features, and let F, F' and every F_i ($1 \leq i \leq n$) implement P. If the pipeline in Figure 5.9(a) does not satisfy a desired safety property, neither does the pipeline in Figure 5.9(b).*

Proof:

Let X_1 be the pipeline segment from F_1 to F_{i-1}, X_2 be the segment from F_i to F_j, and X_3 be the segment from F_{j+1} to F_n in Figure 5.9(b). Suppose X is the pipeline obtained by replacing each X_1, X_2 and X_3 in Figure 5.9(b) with P, i.e., X consists of F, F' and three instances of P. By Lemma 2, if X is not safe, neither is the pipeline in Figure 5.9(b). By Lemma 1 and Theorem 2.3.2 in Section 2.3.2, if the pipeline in Figure 5.9(a) is not safe, neither is X. □

In Section 5.5, we use Theorem 5.4.1 to propose an efficient pipeline ordering algorithm. Another application of this theorem, which we do not consider in this paper, is for pipeline verification. Specifically, it follows from the contrapositive of the theorem that if a given pipeline satisfies a safety property, any subsequence of the pipeline satisfies that property as well.

5.5 Compositional Synthesis

In this section, we describe the algorithm for computing ordering of features in a pipeline, to ensure that they do not admit any of the undesirable interactions. The algorithm, OrderPipeline, is shown in Figure 5.10. The main engine of this algorithm is the function FindPairwiseConstraints, shown in Figure 5.11, which computes a set $\mathcal{C}$ of ordering constraints between feature pairs. These constraints are inferred by model checking the two possible compositions of each feature pair against the safety properties defined over that pair. For example, let $F_1 = \text{CB}$ and $F_2 = \text{RVM}$, and let $\mathsf{negTr} = NS_1$ (see Section 5.2). With these inputs, FindPairwiseConstraints yields CB $<$ RVM because the property NS_1 holds in the composition where CB comes before RVM (line 5 in Figure 5.11), but not in the other composition (line 7). The resulting constraint CB $<$ RVM is added to $\mathcal{C}$ (line 11) which is returned on line 16. By Theorem 5.4.1, a pipeline ordering that does not respect pairwise ordering constraints is unsafe, and thus inadmissible. This provides us with an effective strategy for pruning the search space for solutions.

Given a pair of features, FindPairwiseConstraints can infer an ordering over the pair, if exactly one of their two possible compositions violates the given properties. Otherwise, if neither composition violates the properties, the features in question can be put in any order, and hence no constraint is derived (line 15). If both compositions violate the properties, FindPairwiseConstraints returns error (line 9). In this case, the given features need to be revised before they can be put together in a pipeline; hence, OrderPipeline terminates unsuccessfully (line 3).

If FindPairwiseConstraints does not return error, OrderPipeline enters a repeat-until loop (lines 4–8). Every iteration of this loop starts by finding a permutation of the n features comprising the pipeline that satisfies the set of constraints computed by FindPairwiseConstraints. Such a permutation, called T, satisfies a set $\mathcal{C}$ of constraints if for every constraint $F_k < F_l$ in $\mathcal{C}$, we have $\mathsf{T}[k] < \mathsf{T}[l]$, i.e., feature F_k is

Algorithm. OrderPipeline

Input: - Features $F_1, \ldots, F_n$ with action sets $E_1, \ldots, E_n$, respectively.

- A set $\mathsf{negTr} \subseteq (\bigcup_{1 \leq k \leq n} E_k)^*$ of negative traces.

Output: A permutation, T, of 1 to n giving an order on $F_1, \ldots, F_n$.

1: $\mathcal{C}$:= FindPairwiseConstraints($F_1, \ldots, F_n, \mathsf{negTr}$)

2: **if** ($\mathcal{C} = \mathsf{error}$) **:**

3: **return** error

4: **repeat**

5: T:= Next permutation of $1, 2, \cdots, n$ satisfying $\mathcal{C}$

6: Let B_i bind $F_{\mathsf{T}[i]}.\mathsf{r}$ to $F_{\mathsf{T}[i+1]}.\mathsf{l}$ for $1 \leq i < n$

// B_i connects the feature at position i to the one at position $i+1$

7: safe := ModelCheck($F_{\mathsf{T}[1]} ||_{B_1} \ldots ||_{B_{n-1}} F_{\mathsf{T}[n]}$, negTr)

8: **until** safe

9: **return** T

Figure 5.10: Algorithm for pipeline ordering.

positioned to the left of feature F_l in the pipeline. For example, let F_1 = CB, F_2 = QT, and F_3 = RVM. The permutation T satisfying constraints { CB < RVM, RVM < QT} is $[1, 3, 2]$. Afterwards, a global composition of the features is built with respect to the computed permutation T. If this composition satisfies all the given properties, T is returned as a solution. Otherwise, the loop continues until a solution is found, or all permutations that satisfy $\mathcal{C}$ are exhausted. In the latter case, OrderPipeline returns error.

Notice that merely satisfying $\mathcal{C}$ does not make a given permutation T a solution to the

Algorithm. FINDPAIRWISECONSTRAINTS

Input: Features $F_1, \ldots, F_n$ and negative trace negTr.

Output: A set $\mathcal{C}$ or pairwise ordering constraints.

1: $\mathcal{C} := \emptyset$

2: **for** $1 \leq k < l \leq n$**:** *// choose a pair F_k, F_l*

3: $\mathsf{negTr}' := \mathsf{negTr} \cap (E_k \cup E_l)^*$ *// restrict negTr to F_k and F_l*

4: $B_1 := \text{Binding}(F_k.\mathsf{r}, F_l.\mathsf{l})$ *// put F_k before F_l*

5: $\mathsf{safe}_1 := \text{MODELCHECK}(F_k||_{B_1}F_l, \mathsf{negTr}')$

6: $B_2 := \text{Binding}(F_l.\mathsf{r}, F_k.\mathsf{l})$ *// put F_k after F_l*

7: $\mathsf{safe}_2 := \text{MODELCHECK}(F_k||_{B_2}F_l, \mathsf{negTr}')$

8: **if** $(\neg\mathsf{safe}_1 \wedge \neg\mathsf{safe}_2)$ **:**

9: **return** error

10: **if** $(\mathsf{safe}_1 \wedge \neg\mathsf{safe}_2)$ **:**

11: add $F_k < F_l$ to $\mathcal{C}$

12: **else if** $(\neg\mathsf{safe}_1 \wedge \mathsf{safe}_2)$ **:**

13: add $F_l < F_k$ to $\mathcal{C}$

14: **else :** *// i.e., $\mathsf{safe}_1 \wedge \mathsf{safe}_2$*

15: do nothing *// inconclusive result; no constraint on F_k w.r.t. F_l*

16: **return** $\mathcal{C}$

Figure 5.11: Algorithm for finding pairwise ordering constraints.

pipeline ordering problem. For example, consider features A and B in Figure 5.12(a)[3].

[3]This example is similar to that given in Section 5.2, but the details are not identical.

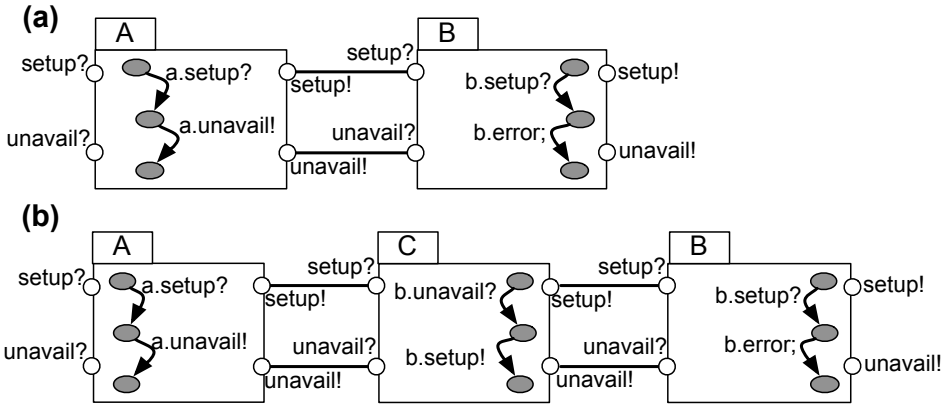


Figure 5.12: Local ordering vs. global ordering.

The composition of A and B in the figure is safe for the trace "b.error;", i.e., "the error action is unreachable". However, once feature C is inserted between A and B in Figure 5.12(b), the resulting pipeline is no longer safe for this property: Theorem 5.4.1 only guarantees safety *violations* to lift from a pairwise to the global setting. However, safety properties that are *satisfied* over a pair of features are not necessarily lifted[4]. Therefore, we need to check all safety properties over the global composition induced by a candidate ordering. Further, although in practice most safety property traces are expressed over pairs of features, we can envision traces that refer to several and potentially to all features in the system. Checking such properties requires the construction of a global composition.

Our pipeline ordering algorithm is *sound* because we construct a global composition and verify it against all the given properties. The algorithm is *complete* because by Theorem 5.4.1, it never prunes an ordering that is a possible solution to the pipeline ordering problem. Finally, the algorithm is *change-aware*, allowing for the reuse of synthesis results across changes to pipelines. Specifically, after adding or modifying a feature F, we

[4]The features in Figure 5.12 can be completed to implement the transparency pattern and yet exhibit the same problem.

can reuse the pairwise constraints that do not involve F, and thus, we do not need to reconsider all possible configurations after each change.

The scalability and effectiveness of our approach ultimately depend on how well we can narrow down the search for potentially admissible pipeline permutations, and whether verifying compositions (lines 5 and 7 in Figure 5.11, and line 7 in Figure 5.10) is feasible. In Section 5.7, we apply our approach to an industrial telecom example. There, we demonstrate that substantial pruning of the search space can be achieved by utilizing the pairwise constraints inferred from the known undesirable interactions in the domain. The features used in our evaluation were not very large, and therefore, we could verify their compositions in a conventional way. But, for larger systems, we can improve the scalability of OrderPipeline algorithm using existing automated compositional techniques for checking safety properties (e.g., (Cobleigh *et al.*, 2003)).

5.6 Implementation

We have developed a prototype implementation of the pipeline ordering algorithm described in Section 5.5. We discuss inputs to the algorithm as well as the relevant technical details below.

5.6.1 Inputs

Our algorithm in Section 5.5 receives a set of features expressed as I/O automata and a set of negative traces capturing undesirable interactions between these features. In order to use standard verification tools, in our case, the LTS Analyzer (LTSA) tool (Magee & Kramer, 2006), our tool translates the input features to LTSs and the negative traces – to property LTSs (see Section 2.2.2).

5.6.2 Parallel composition

Our technique requires us to compute compositions of pipeline features (lines 5 and 7 of FindPairwiseConstraints in Figure 5.11 and line 7 of OrderPipeline in Figure 5.10), for which we need to implement the parallel composition operator $||_B$ (Definition 5.3.4) – one is not readily available in LTSA. This is achieved as follows: first, we do an action relabelling to ensure that shared actions, with respect to a given binding B, have identical labels in the features to be composed. We then apply LTSA's parallel composition operator (Definition 2.2.3) to compose the features.

5.6.3 Model checking

Since we translate negative traces to safety LTSs, model checking (lines 5 and 7 of FindPairwiseConstraints and line 7 of OrderPipeline) can be done directly using LTSA. Note that our technique involves model checking not only pairwise but also the global composition (line 7 of OrderPipeline). Our tool currently uses LTSA directly for this latter check, which has not presented a challenge so far because the number and the size of features we have been working with so far have been relatively small (see Section 5.7). However, this check may become an issue when analyzing larger systems, and in the future we intend to use an enhanced version of LTSA (Cobleigh *et al.*, 2003) that enables compositional model checking for safety properties. This approach applies to our work directly, since the negative traces we use are safety properties.

5.6.4 Ordering permutations

To generate ordering permutations that satisfy a given set of constraints (line 5 of OrderPipeline), we use a backtracking constraint solver, Choco (Laburthe & Jussien, 2008). All constraints used in our approach are binary, and for those, the state-of-the-art look-ahead techniques for solving CSP problems are very efficient.

5.7 Evaluation

In this section, we provide initial evidence for the usefulness of our approach through a case study from the telecom domain. Our study involves six features from AT&T deployed in the DFC architecture (Jackson & Zave, 1998).

When conducting the study, we had a number of goals. The first goal was to check that the features present in the case study simulate our formalization of the transparency pattern in Figure 5.8 (**G1**). The other two goals were to investigate whether our technique can sufficiently narrow down the search for a safe pipeline ordering, which includes the ability to identify enough negative scenarios of interaction (**G2**), and to evaluate the performance of our technique on a realistic example (**G3**). We begin this section with a description of the domain of our study, and discuss the experience with the above goals in Section 5.7.2.

5.7.1 Domain Description

In DFC, a simple telecom usage is implemented by a linear pipeline such as the one shown in Figure 5.1. The original DFC pipeline has several additional signals, e.g., avail and unknown, which we omitted from Figure 5.1 for simplicity. The pipeline in the figure includes six features, namely, CB and RVM (see Section 5.2), as well as QT, SFM, AC, and NATO. A high-level description of the four new features, taken from (Zave, 1999), is as follows:

Quiet Time (QT) enables the subscriber to avoid an incoming call by activating a dialog with the caller, saying that the subscriber wishes not to be disturbed. If the caller indicates that the call is urgent, this feature allows the call to go through. Otherwise, it signals failure (unavail) upstream.

Sequential Find Me (SFM) attempts to find the callee at a sequence of locations. If the first location does not succeed, then while all the other locations are being

tried, the feature plays an announcement, letting the caller know that the call is still active.

Answer Confirm (AC) uses a media resource to elicit confirmation that the call has been answered by a person rather than by a machine. If the test is not passed, it signals unavail upstream, even though the call was actually answered.

No Answer Time Out (NATO) signals failure (unavail) upstream if an incoming call is not answered after a certain amount of time.

The DFC architecture supports dynamic architectural reconfiguration. This means that features and bindings can be created, destroyed, or reassigned at runtime. In fact, the pipeline in Figure 5.1 is a static snapshot of a dynamic structure. For example, in the figure, each new location tried by SFM results in a new setup signal sent downstream, and creation of new instances of AC and NATO. We do not consider such advanced capabilities here. Specifically, we abstract away feature behaviours involving runtime reconfiguration. Hence, a pipeline ordering synthesized by our technique is over a static snapshot of a (potentially) dynamic DFC pipeline. In this sense, the real value of our technique with respect to DFC is as an exploration tool through which analysts can consider different snapshots of the same pipeline and ensure that the synthesized orderings for these snapshots are consistent with one another.

The features in our case study are specified in Boxtalk (Zave & Jackson, 2002) – a domain-specific language for specifying telecom features. Each Boxtalk specification is a state machine with a set of states and a set of transitions which can be triggered by actions. Boxtalk also provides constructs for manipulating data and media, but we do not consider these constructs in this work. Boxtalk is similar to I/O automata in that the models described in it are input-enabled; the language also distinguishes between input, output, and internal actions of features (Zave & Jackson, 2002). Hence, the control behaviours of Boxtalk specifications can be conveniently captured using our I/O

Feature	CB	RVM	QT	SFM	NATO	AC
# of states	9	10	12	22	7	10
# of transitions	13	13	19	31	16	21

Table 5.1: Sizes of the resulting translations.

automata-based formalism (Definition 5.3.2).

In this case study, all of the features except NATO and CB have additional ports through which they communicate with media resources that record speech, play announcements, detect touch-tones, etc. We have abstracted away from these ports, replacing their signals with internal actions such as "rvm.voicemail;". This abstraction is safe because the interaction of each feature with its media resource is independent and and logically contained within the feature, thus not affecting feature composition.

5.7.2 Experience

We manually translated the six Boxtalk features into I/O automata. The sizes of the translated models are shown in Table 5.1, whereas the original Boxtalk specifications and the resulting I/O automata are available in Appendix B.

Our analysis indicates that all these features implement our formalization of the transparency pattern. We already exemplified the simulation relation for CB and RVM in Section 5.4. For the remaining features, see Appendix B. To show that a state machine satisfies the transparency pattern, we need to prove the existence of a stuttering simulation relation between that state machine and the pattern. There are several tools that can check the existence of such relations, such as MAGIC (Chaki *et al.*, 2003). The realization of the transparency pattern satisfies goal **G1** and enables the application of our pipeline ordering algorithm.

G2. The scenarios used in our study are shown in Table 5.2 (left column). These

scenarios came from (Zave, 1999) and from the experience of the domain expert. Note that these scenarios may not be always known in advance. To elicit them, the domain expert may have to inspect or monitor the models and their interactions using automated analysis tools. Table 5.2 (right column) shows the constraints inferred by our technique for the individual scenarios. These constraints were sufficient to conclusively order all the features in Figure 5.1 except for the SFM feature. The role of SFM is to transform a number that was dialed, i.e., a personal number, into some device number: a home phone, a cell phone, etc. Scenarios involving SFM cannot be expressed as sequences of actions because they refer to data, i.e., personal and device numbers. In this work, we do not model data and instead rely on the domain expert to provide the constraints for SFM. Specifically, CB, RVM, and QT should precede SFM because they are personal features, i.e., they apply to the personal number. In contrast, AC and NATO should follow SFM because they apply to each phone try individually, and there will be a different instance of AC and NATO for each try. Using these additional constraints, we were able to narrow down the set of possible global orderings to a single one.

While we had no problem in this domain where the nature of interactions between feature pairs was well studied and well understood, our technique may be less effective in other domains. The degree to which it narrows down the search is influenced by factors such as the size and the number of features in the domain, the amount of domain expertise available, and the existence of formal design guidelines for feature development, and all of these may vary widely.

To extend the applicability of our approach to domains where an adequate set of negative scenarios is hard to obtain, the approach can be combined with simulation and monitoring tools which assist users in identifying additional undesirable scenarios. The idea is that analysts often have certain heuristics for detecting "suspicious" behaviour, even though they may not have pinned down the exact undesirable interactions. For example, it might be dangerous for certain pairs of features to be active in the same usage

Negative Scenario	**Constraint(s)**
QT cannot stop a caller from leaving a voicemail message.	RVM < QT
A blocked caller should not be allowed to engage in a dialogue with the system (this is to avoid wasting expensive media resources).	CB < AC, CB < QT CB < RVM
If QT is enabled and the call is not urgent, the system should not disturb the callee with a confirmation dialogue.	QT < AC
The timer interval should never include the time that the system takes having a dialogue with a user (because that should not be included in the time allowance for answering).	AC < NATO, QT < NATO

Table 5.2: Negative scenarios and the resulting constraints.

scenario. The ability of a tool to report pairs of features that can be active simultaneously may help analysts to identify additional safety properties and thus reduce the number of feature orderings.

Different monitoring tools can be used in conjunction with our approach, but the one that readily integrates with our formalism is LTSA's simulation module. This module can be used to monitor the parallel composition of a set of features and report traces leading to suspicious behaviours. These traces can then be studied by analysts as potential candidates for negative scenarios. Since our approach requires traces only over pairs of features to infer ordering constraints, users can concentrate on pairwise compositions, for which traces are typically small and intuitive enough for manual inspection.

G3. We measured the time and memory performance of the different steps in our technique, applied to the features in our study. The reported times are for a PC with a 2.2GHz Pentium Core Duo CPU and 2GB of memory; our implementation used version 1.2 of Choco and version 2.3 of LTSA.

FindPairwiseConstraints: Executing lines 5 and 7 of this algorithm (Figure 5.11)

involves building pairwise compositions of LTSs and model checking them. Since I/O automata can be seen as LTSs, the sizes of our LTS translations are those shown in Table 5.1. The number of states of the pairwise compositions ranged between 60 to 259, and the number of transitions between 210 to 785. The running times for generating the compositions were negligible, i.e., under 1s.

To model check the compositions, we expressed safety properties as (safety) LTSs, which, for the properties in Table 5.2, ranged between 3 to 5 states, and 5 to 8 transitions. For example, Figure 2.4(a) can be interpreted as a safety LTS for the property NS_1 described in Section 5.2 by letting $a =$ "cb.reject;" and $b =$ "rvm.voicemail". The running times of individual model checking tasks were negligible.

For the six features in the study and the properties in Table 5.2, the total execution time of FindPairwiseConstraints was 6.47s and the maximum required memory was 10M. The result of running the algorithm is the set of ordering constraints in the second column of Table 5.2.

OrderPipeline: Line 5 of this algorithm (Figure 5.10) invokes a constraint solver Choco to compute a permutation satisfying the pairwise ordering constraints. The running time and memory usage of this step were negligible due to the nature of our CSP problem (see Section 5.6), and resulted in a single permutation that satisfied all of the pairwise constraints in Table 5.2.

Line 7 of the OrderPipeline algorithm requires computing a global composition of the features. Since there is only one permutation satisfying the constraints in Table 5.2, only one global composition needed to be built and verified. The number of states and transitions in this global composition are 1.5×10^6 and 22×10^6, respectively[5]. The time and memory needed for generating this composition are 71.4s and 913M, respectively. The total model checking time, i.e., the sum of model checking times for individual

[5]We have observed that global compositions for other permutations are roughly of the same size.

properties in Table 5.2, was 16min, and the maximum memory requirement was 1G.

Overall, we were able to compute a safe feature ordering in about a quarter of an hour. The order that we computed is the same as the one that was produced by the domain expert via manual analysis of the feature pairs. As we discussed in Section 5.2, this may not be the case when the features do not satisfy the transparency pattern. The most expensive part of our algorithm is model checking of a global composition, which took about 16min. This cost is incurred no matter what approach one takes for ordering a set of features. Even if we were to select a feature ordering randomly, we would still have to build the global composition and verify it. Since the size of global compositions grows quickly, compositional techniques for dealing with space explosion are needed. While we managed to build global compositions using LTSA in our case study without resorting to compositional analysis tools, efficient tools for checking global compositions already exist and can be readily incorporated into our approach as discussed in Section 5.6.

5.8 Related Work

The ideas and techniques presented in this chapter are related to the following threads of research: feature interaction analysis, compositional reasoning, and design for verification approaches.

5.8.1 Feature interaction

Feature interaction is a well studied problem in feature-based software development, and many approaches addressing this problem have been proposed, e.g., (Jackson & Zave, 1998; Blom *et al.*, 1994; Hay & Atlee, 2000; Plath & Ryan, 2001; Hall, 2000; Li *et al.*, 2002). Some of the earlier work required extensive manual intervention. For example, (Blom *et al.*, 1994) proposed a logic-based approach for detecting undesirable interactions by manually instrumenting potential interactions with exception clauses and employing

a theorem prover for finding inconsistencies. (Plath & Ryan, 2001) developed a scheme whereby users manually specify the join points at which new features can be inserted into a base system. Feature interactions are explored by weaving the features and model checking the result.

Recent approaches offer better automation: (Hay & Atlee, 2000) resolves undesirable interactions using predefined priorities prescribing which feature should be favoured should a conflict arise. (Li *et al.*, 2002) proposes a compositional method for verifying features that are composed sequentially through known interface states. (Hall, 2000) provides an automated unification operator for combining features (described in a functional language) with respect to given unifiers. These approaches assume that the relationships between features (i.e., priorities in (Hay & Atlee, 2000), interfaces in (Li *et al.*, 2002), and unifiers in (Hall, 2000)) are developed *a priori*. Our work deals with a complimentary problem of how to *synthesize* these relationships.

Several approaches propose the use of architectural styles, such as layered (Brooks, 1986; Pomakis & Atlee, 1996) or pipeline architectures (Jackson & Zave, 1998; Braithwaite & Atlee, 1994), as a way to prevent undesirable interactions. The arrangement of features within these architectures is typically done manually. Our work offers a solution to automate this task for the pipeline architectural style.

Our work also relates to (Dominguez & Day, 2005) which focuses on verifying port-based systems (with buffered links between features). The work builds on domain-specific knowledge about the DFC architecture, such as regularity of feature properties and symmetry of communication port behaviours, to enable compositional verification of DFC usage pipelines. However, (Dominguez & Day, 2005) does not address the synthesis of these pipelines, and also does not exploit the transparent behaviours of DFC features for reasoning.

The closest work to ours is that of (Zimmer, 2007) which reduces the effort of computing global feature orderings by partitioning features into categories, and then separately

sorting the set of categories and the set of features found within each category. To make their analysis scalable, the authors use an assumption similar to transparency which is that features in a scenario can be arbitrarily added or removed at runtime. Reasoning about soundness in (Zimmer, 2007) is similar to ours; however, since the authors do not formalize transparency, they cannot reason about the completeness of their approach. In contrast, our formalization of feature behaviours and the notion of transparency enables us to show completeness in addition to soundness.

5.8.2 Compositional analysis

Compositional analysis extends the applicability of verification methods by reducing reasoning about a large system to reasoning about its individual components. For example, assume-guarantee reasoning (Pnueli, 1985) enables verification of individual components in conjunction with assumptions about their environment, and allows lifting of the results to the entire system. Existing work on this topic (Cheung & Kramer, 1996) is not directly applicable to synthesis of feature-based systems because one needs to know about the bindings between features in order to specify the environmental assumptions. However, our synthesis algorithms could directly benefit from *automated* assume-guarantee techniques for safety properties, e.g., (Cobleigh *et al.*, 2003). Specifically, our feature ordering algorithm in Section 5.5 involves model checking pairwise and global compositions induced by specific bindings. These model checking tasks can be done more efficiently using the technique of (Cobleigh *et al.*, 2003).

5.8.3 Design for verification

Design for verification promotes the use of domain-specific patterns and guidelines to facilitate efficient automated verification (Sharygina *et al.*, 2001; Mehlitz & Penix, 2005; Betin-Can *et al.*, 2005; Cheng *et al.*, 2005). For example, (Betin-Can *et al.*, 2005) provides a concurrency controller pattern for making verification of concurrent Java pro-

grams more scalable, and (Sharygina *et al.*, 2001) studies the use of structural design guidelines for efficient verification of UML models. To the best of our knowledge, the use of patterns for compositional reasoning about feature-based systems has not been investigated previously.

5.9 Limitations

In addition to the limitations discussed in Section 5.7.1 regarding the case study from the DFC domain, we have made two assumptions in our work which may limit the generalizability of our approach. First, our formalism, i.e., I/O automata, captures only control behaviours of features and cannot model their data dependencies. As a result, our analysis does not address the feature interactions that may occur due to the interplay between data values and feature behaviours. Second, our result is applicable to cases where features have identical interfaces, and hence, can get glued together in any arbitrary order. This may not necessarily be a common case in feature-based development, particularly when features have distinct interfaces that can significantly constrain the number of ways in which features can be put together.

5.10 Conclusion

In this chapter, we presented a sound and complete compositional approach for synthesizing pipeline feature orderings. The formal groundwork for our technique is a pattern of behaviour called transparency. We proved that this pattern enables inferring global constraints on feature arrangements through pairwise analysis of the features. We reported on a prototype implementation and preliminary evaluation of our work for synthesizing orderings of AT&T telecom features.

In our case study, the desired properties were described as negative traces (safety). Our algorithm can also readily work with positive traces. In fact, a corollary of Theo-

rem 5.4.1 is that the transparency pattern lifts existence of positive traces from a pairwise to the global setting. We leave extending our technique to (finite) liveness, i.e., dealing with *universal* positive behaviours, to future work.

Further, the properties in our case study did not contradict one another. However, one could envision cases where some of these properties are mutually inconsistent. Our feature ordering algorithm detects such inconsistencies as circular, or over-constrained, orderings. Alternatively, users may want to establish consistency before applying our algorithm, e.g., using Alloy (Jackson, 2006).

While this chapter focused on pipeline arrangements, our technique can be used for synthesis of more complex arrangements as well. In particular, we can synthesize graph arrangements consisting of linear pipeline segments by first synthesizing the segments and then combining them to construct the overall system.

Our current approach assumes that each feature implements a distinct requirement. Yet, it is possible for individual features, particularly in legacy systems, to implement multiple functions, each invoked depending on external parameters or user preferences. Our approach may create circular dependencies between such features. We leave the methodology of breaking these cycles and dealing with parameterized feature arrangements for future work.

Chapter 6

Conclusion

In this chapter, we summarize the contributions of this thesis and outline directions for future research.

6.1 Summary of The Thesis

We studied three instances of the fusion problem for behavioural models:

Merging complementary models. In Chapter 3, we studied state machines describing complementary perspectives on a single feature of a system. We used 3-valued logic to explicitly specify the incomplete behaviours of the state machine models. We provided an algorithm for computing a consistency relation over the states of the 3-valued state machines, i.e., a relation that maps states satisfying the same temporal properties. Using this relation, we constructed a merge that preserves the temporal properties of the original models.

Merging variant implementations of the same system. In Chapter 4, we studied state machines describing variant specifications of individual system features. The goal here was to merge the variants while preserving their points of difference. We formalized variants as parameterized state machines to explicitly distinguish

between common and variable behaviours. We developed heuristic techniques to identify commonalities and variabilities between model variants, and computed their merge based on the notion of common refinement defined over parameterized state machines.

Composing features and analysing their interactions. In Chapter 5, we studied the problem of interaction analysis over an evolving set of state machines that describe different features of a system. We formalized features as I/O automata which distinguish the input, internal, and output actions of each feature. We proposed a technique for verifying compositions of features arranged in a pipeline architecture. To make our technique efficient for systems whose features periodically change, we identified a pattern of behaviour which enables for the re-use of verification results across changes to the pipeline features.

The key observations driving the research in this thesis are the following: (1) models are often partial, inconsistent, and open; (2) to be able to combine models, we need to make all assumptions about the relationships between the models explicit; and (3) the process of combining a set of inter-related models depends on the nature of the relationships between the models and the intended applications of the models.

The effectiveness of a solution to a model fusion problem depends primarily on (1) the expressive power of the formalisms used to capture models and their relationships; (2) the level of automation that the solution provides; and (3) the scalability of the solution. In this thesis, we used well-studied and generalizable formalisms to capture models and relationships. In each case, we discussed the aspects that our formalisms can and cannot express. We provided automated algorithms for a number of important operations, e.g., computing consistency relations in Chapter 3, merging in Chapters 3 and 4, and model checking feature compositions in Chapter 5. In addition, we identified other equally important operations that cannot be fully automated, e.g., matching in Chapter 4, and

eliciting undesirable interaction scenarios in Chapter 5. For these non-automatizable operations, we discussed heuristics and tool-supported techniques to assist developers in performing the operations. Finally, we evaluated our solutions on medium-sized case studies.

6.2 Future Directions for the Thesis

In this section, we outline a list of challenges that we faced in our work, providing suggestions for future research in the model fusion area.

6.2.1 Relationships between Models

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

These operators can be improved in a number of ways. For example, they can be used interactively, with the developer seeding them with some of the more obvious matches, and pruning incorrect ones iteratively. Or, they can be customized for specific domains using learning-based techniques (Mandelin *et al.*, 2006).

In addition to identifying model relationships, we need to ensure that these relationships are semantically meaningful. One way to achieve this is to first compute the merge of the given models with respect to their relationships, and then apply automated analyses, e.g., consistency checking, to ensure that the relationships between models result in a merge which satisfies the properties of interest (Sabetzadeh *et al.*, 2007b). In situations

where merges are very large, we may investigate compositional techniques to reduce reasoning about the merge to reasoning about smaller subsets of models and relationships.

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

6.2.2 Heterogeneous Merge

Heterogeneous merge is often carried out using transformations and manipulations defined at the meta-model level (Bezivin *et al.*, 2005). Meta-model level transformations, despite being general and flexible, typically deal only with syntactic and visual aspects of models. To generate merges that are semantically sound and to better mechanize the matching process, we could define meta-models that are more than just the abstract syntax of a language, e.g., by augmenting meta-model languages with logical constraints or behavioural specification languages such as activity and sequence models (France & Rumpe, 2007).

6.2.3 Tool Support

Many industrial distributed and collaborative model-based development tools include some support for model merging. A careful examination of the model merging processes in these tools reveals a number of important shortcomings. In particular, most existing industrial modelling platforms, e.g., the Rational Software Architect (Letkeman, 2006), are primarily aimed at centralized development, where all developers contribute to a single holistic model. Fragments of this model are visualized as views containing diagrams

(potentially) in different notations, e.g., class or sequence diagrams. These tools lack support for merging independently developed models as they often do not allow developers to explicitly construct relationships between models. In these tools, elements in different views are considered similar only if they are different copies of the same element in the holistic model. This is inadequate for merging independently developed models where elements in different models may be similar due to their syntactic and semantic characteristics.

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

6.2.4 Verification of Collections of Inter-related Models

Fusion tasks are often intertwined with verification tasks to ensure that the manipulations performed over models preserve their well-formedness and desired semantic properties. For example, in Chapter 5, we employ model checking to verify that the composition of a given set of features satisfies the desirable properties. Similarly, (Sabetzadeh *et al.*, 2007b) combines model merging and (intra-model) consistency checking to enable construction of sound and meaningful relationships between a set of models. To build and manage evolving systems of interrelated models, we need to devise scalable verification techniques that (1) are robust with respect to system changes, and (2) can check not only classical properties of models, but also non-classical ones, such as those involving

inconsistent and incomplete aspects of models.

6.2.5 Software Reliability

Use of models provides an opportunity to design software systems with reliability in mind rather than analyzing and testing software code for reliability after the fact. One approach to building software systems that are reliable by design and more amenable to scalable verification is to use generic or domain-specific guidelines and patterns for constructing software models (Betin-Can *et al.*, 2005). For example, in Chapter 5, we provided a pattern of behaviour for pipeline features that allows for synthesis of reliable feature compositions. This work can be extended by applying the current technique to dynamically reconfigurable systems, i.e., systems whose features and architectural links can change at run-time. This extension would enable efficient construction of reliable software systems with support for self-management and adaptation.

References

Abadi, M., & Lamport, L. 1993. "Composing Specifications". *ACM Transactions on Programming Languages and Systems*, **15**(1), 73–132.

Alanen, M., & Porres, I. 2003. "Difference and Union of Models". *Pages 2–17 of: UML '03: Proceedings of the 6th International Conference on The Unified Modeling Language*. Lecture Notes in Computer Science, vol. 2863. Springer.

Alspaugh, T., Antón, A., Barnes, T., & Mott, B. 1999. "An Integrated Scenario Management Strategy". *Pages 142–149 of: RE '99: Proceedings of the 4th IEEE International Symposium on Requirements Engineering*.

Alur, R., Henzinger, T., & Kupferman, O. 1997. "Alternating-time Temporal Logic". *Pages 100–109 of: FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science*.

Alur, R., Kannan, S., & Yannakakis, M. 1999. "Communicating Hierarchical State Machines". *Pages 169–178 of: ICALP '99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 1644. Springer.

Antonik, A., Huth, M., Larsen, K., Nyman, U., & Wasowski, A. 2008. "Complexity of Decision Problems for Mixed and Modal Specifications". *Pages 112–126 of: FoSSaCS '08: Proceedings of the 11th International Conference on Foundations of Software*

Science and Computational Structures. Lecture Notes in Computer Science, vol. 4962. Springer.

Balcázar, J., Gabarró, J., & Santha, M. 1992. "Deciding Bisimilarity is P-Complete". *Formal Aspects of Computing*, **4**(6A), 638–648.

Batory, D. 2004. "Feature-Oriented Programming and the AHEAD Tool Suite". *Pages 702–703 of: ICSE '04: Proceedings of the 26th International Conference on Software Engineering*.

Belnap, N. 1977. "A Useful Four-Valued Logic". *Pages 5–37 of: Modern Uses of Multiple-Valued Logic*. Reidel.

Bernstein, P. 2003. "Applying Model Management to Classical Meta Data Problems". *Pages 209–220 of: Proceedings of the 1st Biennial Conference on Innovative Data Systems Research*.

Bernstein, P., Melnik, S., Petropoulos, M., & Quix, C. 2004. "Industrial-Strength Schema Matching". *SIGMOD Record*, **33**(4), 38–43.

Betin-Can, A., Bultan, T., Lindvall, M., Lux, B., & Topp, S. 2005. "Application of Design for Verification with Concurrency Controllers to Air Traffic Control Software". *Pages 14–23 of: ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*.

Bezivin, J., Jouault, F., & Touzet, D. 2005. *"An Introduction to the ATLAS Model Management Architecture"*. Tech. rept. 05-01. LINA.

Blom, J., Jonsson, B., & Kempe, L. 1994. "Using Temporal Logic for Modular Specification of Telephone Services". *Pages 197–216 of: FIW '94: Proceedings of Feature Interactions in Telecommunications Systems*.

Bond, G. 2006. *"An Introduction to ECharts: The Concise User Manual"*. Tech. rept. AT&T. Available at: `http://echarts.org`.

Bond, G., & Goguen, H. 2002. *"ECharts: Balancing Design and Implementation"*. Tech. rept. AT&T. Available at: `http://echarts.org`.

Bond, G., Cheung, E., Purdy, K., Zave, P., & Ramming, J. 2004. "An Open Architecture for Next-Generation Telecommunication Services". *ACM Transactions on Internet Technology*, **4**(1), 83–123.

Braithwaite, K., & Atlee, J. 1994. "Towards Automated Detection of Feature Interactions". *Pages 36–59 of: FIW '94: Proceedings of Feature Interactions in Telecommunications Systems.*

Brand, D., & Zafiropulo, P. 1983. "On Communicating Finite-State Machines". *Journal of the ACM*, **30**(2), 323–342.

Brooks, R. 1986. "A Robust Layered Control System for a Mobile Robot". *IEEE Journal of Robotics and Automation*, 2–27.

Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., & Sabetzadeh, M. 2006. "A Manifesto for Model Merging". *In: Workshop on Global Integrated Model Management (GaMMa '06) co-located with ICSE'06.*

Bruns, G. 2005. "Foundations for Features". *Pages 3–11 of: FIW '05: Proceedings of Feature Interactions in Telecommunications Systems.*

Bruns, G., & Godefroid, P. 2000. "Generalized Model Checking: Reasoning about Partial State Spaces". *Pages 168–182 of: CONCUR '00: Proceedings of 18th International Conference on Concurrency Theory.* Lecture Notes in Computer Science, vol. 1877. Springer.

Chaki, S., Clarke, E., Groce, A., Jha, S., & Veith, H. 2003. "Modular verification of software components in C". *Pages 385–395 of: ICSE '03: Proceedings of the 25th International Conference on Software Engineering.*

Chechik, M., Devereux, B., Easterbrook, S., & Gurfinkel, A. 2003. "Multi-Valued Symbolic Model-Checking". *ACM Transactions on Software Engineering and Methodology*, **12**(4), 371–408.

Cheng, B., Stephenson, R., & Berenbach, B. 2005. "Lessons Learned from Automated Analysis of Industrial UML Class Models (An Experience Report)". *Pages 324–338 of: MoDELS '05: Proceedings of the 8th International Conference Model Driven Engineering Languages and Systems.* Lecture Notes in Computer Science, vol. 3713. Springer.

Cheung, SC, & Kramer, J. 1996. "Context constraints for compositional reachability analysis". *ACM Transaction on Software Engineering Methodologies*, **5**(4), 334–377.

Clarke, E., Emerson, E., & Sistla, A. 1986. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, **8**(2), 244–263.

Clarke, E., Grumberg, O., & Long, D. 1994. "Model Checking and Abstraction". *ACM Transactions on Programming Languages and Systems*, **19**(2), 1512–1542.

Clarke, E., Wing, J., Alur, R., Cleaveland, R., Dill, D., Emerson, A., Garland, S., German, S., Guttag, J., Hall, A., Henzinger, T., Holzmann, G., Jones, C., Kurshan, R., Leveson, N., McMillan, K., Moore, J., Peled, D., Pnueli, A., Rushby, J., Shankar, N., Sifakis, J., Sistla, P., Steffen, B., Wolper, P., Woodcock, J., & Zave, P. 1996. "Formal Methods: State of the Art and Future Directions". *ACM Computing Surveys*, **28**(4), 626–643.

Clarke, E., Grumberg, O., & Peled, D. 1999. *Model Checking.* MIT Press.

Cleaveland, R., Iyer, S. Purushothaman, & Yankelevich, D. 1995. "Optimality in Abstractions of Model Checking". *Pages 51–63 of: SAS '95: Proceedings of the 2nd International Static Analysis Symposium.* Lecture Notes in Computer Science, vol. 983. Springer.

Cobleigh, J., Giannakopoulou, D., & Pasareanu, C. 2003. "Learning Assumptions for Compositional Verification". *Pages 331–346 of: TACAS '03: Proceedings of 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Lecture Notes in Computer Science, vol. 2619. Springer.

Cousot, P., & Cousot, R. 1977. "Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction or Approximation of Fixpoints". *Pages 238–252 of: POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.*

Dams, D., & Namjoshi, K. 2005. "Automata as Abstractions". *Pages 216–232 of: VMCAI '05: Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation.* Lecture Notes in Computer Science, vol. 3385.

Dams, D., Gerth, R., & Grumberg, O. 1997. "Abstract Interpretation of Reactive Systems". *ACM Transactions on Programming Languages and Systems*, **2**(19), 253–291.

de Alfaro, L., & Henzinger, T. 2001. "Interface Automata". *Pages 109–120 of: ESEC/FSE '01: Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering.*

de Alfaro, L., Faella, M., & Stoelinga, M. 2004a. "Linear and Branching Metrics for Quantitative Transition Systems". *Pages 97–109 of: ICALP '04: Proceedings of the 31st International Colloquium on Automata, Languages and Programming.*

de Alfaro, L., Godefroid, P., & Jagadeesan, R. 2004b. "Three-Valued Abstractions of Games: Uncertainty, but with Precision". *Pages 170–179 of: LICS '04: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science.*

Dominguez, A. Juarez, & Day, N. 2005. "Compositional Reasoning for Port-Based Distributed Systems". *Pages 376–379 of: ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering.*

Easterbrook, S., & Chechik, M. 2001. "A Framework for Multi-Valued Reasoning Over Inconsistent Viewpoints". *Pages 411–420 of: ICSE '01: Proceedings of the 23rd International Conference on Software Engineering.*

Emerson, A., & Kahlon, V. 2000. "Reducing Model Checking of the Many to the Few". *Pages 236–254 of: CADE '00: Proceedings of 17th International Conference on Automated Deduction.*

Emerson, A., & Namjoshi, K. 2003. "On Reasoning About Rings". *International Journal on Foundations of Computer Science*, **14**(4), 527–550.

Faulk, S. 2001. "Product-Line Requirements Specification (PRS): An Approach and Case Study". *Pages 48–55 of: RE '01: Proceedings of the 6th IEEE International Symposium on Requirements Engineering.*

Felty, A., & Namjoshi, K. 2003. "Feature Specification and Automated Conflict Detection". *ACM Transactions on Software Engineering and Methodology*, **12**(1), 3–27.

Fischbein, D., & Uchitel, S. 2008. "On Correct and Complete Strong Merging of Partial Behaviour Models". *In: SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* To appear.

Fisler, K., & Krishnamurthi, S. 2005. "Decomposing Verification by Features". *In: VSTTE '05: Proceedings of Verified Software: Theories, Tools, Experiments.*

France, R., & Rumpe, B. 2007. "Model-Driven Development of Complex Software: A Research Roadmap". *Pages 37–55 of: FOSE '07: 2007 Future of Software Engineering.*

Godefroid, P., & Huth, M. 2005. "Model Checking vs. Generalized Model Checking: Semantic Minimizations for Temporal Logics". *Pages 158–167 of: LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science.*

Godefroid, P., & Jagadeesan, R. 2002. "Automatic Abstraction Using Generalized Model-Checking". *Pages 137–150 of: CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification.* Lecture Notes in Computer Science, vol. 2404. Springer.

Godefroid, P., & Jagadeesan, R. 2003. "On the Expressiveness of 3-Valued Models". *Pages 206–222 of: VMCAI '03: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation.* Lecture Notes in Computer Science, vol. 2575. Springer.

Gomaa, H. 2004. *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures.* first edn. Addison Wesley.

Grosu, R., & Smolka, S. 2005. "Safety-Liveness Semantics for UML 2.0 Sequence Diagrams". *Pages 6–14 of: ACSD '05: Proceedings of the Fifth International Conference on Application of Concurrency to System Design.*

Gruber, T. 1991. "The Role of Common Ontology in Achieving Sharable, Reusable Knowledge Bases.". *Pages 601–602 of: Principles of Knowledge Representation and Reasoning.*

Grumberg, O., & Long, D. 1994. "Model Checking and Modular Verification". *ACM Transactions on Programming Languages and Systems*, **16**(3), 843–871.

Gurfinkel, A., & Chechik, M. 2005. "How Thorough is Thorough Enough". *Pages 65–80 of: CHARME '05: Proceedings of 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, vol. 3725.

Gurfinkel, A., & Chechik, M. 2006. "Why Waste a Perfectly Good Abstraction?". *Pages 212–226 of: TACAS '06: Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Lecture Notes in Computer Science, vol. 3920.

Hall, R. 2000. "Feature Combination and Interaction Detection via Foreground/Background Models". *Computer Networks*, **32**(4), 449–469.

Halmans, G., & Pohl, K. 2003. "Communicating the Variability of a Software-Product Family to Customers". *Software and System Modeling*, **2**(1), 15–36.

Harel, D., & Pnueli, A. 1985. "On the Development of Reactive Systems". *Pages 477–498 of: Logics and Models of Concurrent Systems.*

Harel, D., & Politi, M. 1998. *Modeling Reactive Systems With Statecharts : The Statemate Approach.* McGraw Hill.

Harrison, W., Ossher, H., Tarr, P., Kruskal, V., & Tip, F. 2002. *"CAT: A Toolkit for Assembling Concerns".* Tech. rept. RC22686. IBM Research.

Harrison, W., Ossher, H., & Tarr, P. 2006. "General Composition of Software Artifacts". *Pages 194–210 of: 5th International Symposium Software Composition (SC'06), co-located with ETAPS'06.* Lecture Notes in Computer Science, vol. 4089. Springer.

Hay, J., & Atlee, J. 2000. "Composing Features and Resolving Interactions". *Pages 110–119 of: SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering.*

Hayes, J. Huffman, Dekhtyar, A., & Osborne, J. 2003. "Improving Requirements Tracing via Information Retrieval". *Pages 138–147 of: RE '03: Proceedings of the 11th IEEE International Symposium on Requirements Engineering.*

Heitmeyer, C., Bull, A., Gasarch, C., & Labaw, B. 1995. "SCR*: A Toolset for Specifying and Analyzing Requirements". *In: Proceedings of 11th Workshop on Specification of Abstract Data Types Joint with the 8th COMPASS Workshop.*

Horwitz, S., Prins, J., & Reps, T. 1989. "Integrating Noninterfering Versions of Programs". *ACM Transaction on Programming Languages and Systems*, **11**(3), 345–387.

Hussain, A., & Huth, M. 2004. "On Model Checking Multiple Hybrid Views". *Pages 235–242 of: Proceedings of 1st International Symposium on Leveraging Applications of Formal Methods.*

Huth, M., & Pradhan, S. 2001. "Model-Checking View-Based Partial Specifications". *Electronic Notes Theoretical Computer Science*, **45**.

Huth, M., Jagadeesan, R., & Schmidt, D. A. 2001. "Modal Transition Systems: A Foundation for Three-Valued Program Analysis". *Pages 155–169 of: ESOP '01: Proceedings of 10th European Symposium on Programming.* Lecture Notes in Computer Science 2028. Springer.

Jackson, D. 2002. "Alloy: A Lightweight Object Modelling Notation". *ACM Transactions on Software Engineering and Methodology*, **11**(2), 256–290.

Jackson, D. 2006. *Software Abstractions Logic, Language, and Analysis.* The MIT Press.

Jackson, M., & Zave, P. 1998. "Distributed Feature Composition: a Virtual Architecture for Telecommunications Services". *IEEE Transactions on Software Engineering*, **24**(10), 831 –847.

Jaffe, M., Leveson, N., Heimdahl, M., & Melhart, B. 1991. "Software Requirements Analysis for Real-Time Process-Control Systems". *IEEE Transactions on Software Engineering*, **17**(3), 173–182.

Khoumsi, A. 1997. "Detection and Resolution of Interactions between Services of Telephone Networks". *Pages 78–92 of: FIW '97: Feature Interactions in Telecommunications Networks*.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. 2001. "An Overview of AspectJ". *Pages 327–353 of: ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*.

Kleene, S. C. 1952. *Introduction to Metamathematics*. New York: Van Nostrand.

Kozen, D. 1983. "Results on the Propositional μ-calculus". *Theoretical Computer Science*, **27**, 333–354.

Kupferman, O., Vardi, M., & Wolper, P. 2001. "Module Checking". *Information and Computation*, **164**(2), 322–344.

Laburthe, F., & Jussien, N. 2008. *"The Choco Constraint Programming System"*. http://choco-solver.net/.

Larsen, K.G. 1989. "Modal Specifications". *Pages 232–246 of: Automatic Verification Methods for Finite State Systems*. Lecture Notes in Computer Science 407. Springer.

Larsen, K.G., & Thomsen, B. 1988. "A Modal Process Logic". *Pages 203–210 of: LICS '88: Proceedings of 3rd Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press.

Larsen, K.G., Steffen, B., & Weise, C. 1995. "A Constraint Oriented Proof Methodology Based on Modal Transition Systems". *Pages 17–40 of: TACAS '95: Proceedings of First International Workshop on Tools and Algorithms for Construction and Analysis of Systems.* Lecture Notes in Computer Science, vol. 1019. Springer.

Letkeman, K. 2006. *"Comparing and Merging UML Models in IBM Rational Software Architect".* Tech. rept. IBM. http://www-306.ibm.com/software/awdtools/architect/swarchitect/.

Li, H., Krishnamurthi, S., & Fisler, K. 2002. "Verifying Cross-cutting Features as Open Systems". *Pages 89–98 of: SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering.*

Li, H., Krishnamurthi, S., & Fisler, K. 2005. "Modular Verification of Open Features Using Three-Valued Model Checking". *Journal of Automated Software Engineering*, **12**(3), 349–382.

Lynch, N., & Tuttle, M. 1987. "Hierarchical Correctness Proofs for Distributed Algorithms". *Pages 137–151 of: PODC '87: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing.*

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

Maiden, N., & Sutcliffe, A. 1992. "Exploiting Reusable Specifications Through Analogy". *Communications of the ACM*, **35**(4), 55–64.

Mandelin, D., Kimelman, D., & Yellin, D. 2006. "A Bayesian Approach to Diagram Matching with Application to Architectural Models.". *Pages 222–231 of: ICSE '06: Proceedings of the 28th International Conference on Software Engineering.*

Manning, C., & Schütze, H. 1999. *"Foundations of Statistical Natural Language Processing"*. MIT Press.

Mehlitz, P., & Penix, J. 2005. "Design for Verification with Dynamic Assertions". *Pages 285–292 of: SEW '05: Proceedings of the 29th Annual IEEE/NASA on Software Engineering Workshop.*

Mehra, A., Grundy, J. C., & Hosking, J. G. 2005. "A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design". *Pages 204–213 of: ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering.*

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

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

Moreira, A., Rashid, A., & Araújo, J. 2005. "Multi-Dimensional Separation of Concerns in Requirements Engineering". *Pages 285–296 of: RE '05: Proceedings of the 10th IEEE International Symposium on Requirements Engineering.*

Nejati, S. 2005. *"Abstraction in Software Model Checking"*. Tech. rept. CSRG-546. U. of Toronto.

Nejati, S., & Chechik, M. 2005. "Let's Agree to Disagree". *Pages 287 – 290 of: ASE '05: Proceedings of 20th IEEE International Conference on Automated Software Engineering.*

Nejati, S., Gheorghiu, M., & Chechik, M. 2006. "Thorough Checking Revisited". *Pages 106–116 of: FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design.* IEEE Computer Society.

Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., & Zave, P. 2007. "Matching and Merging of Statecharts Specifications". *Pages 54–64 of: ICSE '07: Proceedings of the 29th International Conference on Software Engineering.*

Nejati, S., Sabetzadeh, M., Chechik, M., Uchitel, S., & Zave, P. 2008. "Towards Compositional Synthesis of Evolving Systems". *In: SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* To appear.

Nentwich, C., Emmerich, W., & Finkelstein, A. 2003. "Consistency Management with Repair Actions". *Pages 455–464 of: ICSE '03: Proceedings of the 25 International Conference on Software Engineering.*

Nicola, R. De, Montanari, U., & Vaandrager, F. 1990. "Back and Forth Bisimulations". *Pages 152–165 of: CONCUR '90: Proceedings of 8th International Conference on Concurrency Theory.* Lecture Notes in Computer Science, vol. 458.

Niu, J., Atlee, J. M., & Day, N. A. 2003. "Template Semantics for Model-Based Notations". *IEEE Transactions on Software Engineering*, **29**(10), 866–882.

Ossher, H., & Tarr, P. 2000. "Hyper/J: multi-dimensional separation of concerns for Java". *Pages 734–737 of: ICSE '00: Proceedings of the 22nd International Conference on Software Engineering.*

Pedersen, T., Patwardhan, S., & Michelizzi, J. 2004. "WordNet: Similarity - Measuring the Relatedness of Concepts". *Pages 1024–1025 of: AAAI '04: Proceedings of Association for the Advancement of Artificial Intelligence.*

Plath, M., & Ryan, M. 2001. "Feature Integration Using a Feature Construct". *Science of Computer Programming*, **41**(1), 53–84.

Pnueli, A. 1985. "In Transition from Global to Modular Temporal Reasoning about Programs". *Logic and Models of Concurrent Systems*, 123–144.

Pomakis, K., & Atlee, J. 1996. "Reachability Analysis of Feature Interactions: A Progress Report". *Pages 216–223 of: ISSTA '96: Proceedings of the 1996 International Symposium on Software Testing and Analysis.*

Prehofer, C. 1997. "Feature-Oriented Programming: A Fresh Look at Objects". *Pages 419–443 of: ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming.*

Rahm, E., & Bernstein, P. 2001. "A Survey of Approaches to Automatic Schema Matching". *The VLDB Journal*, **10**(4), 334–350.

Ryan, K., & Mathews, B. 1993. "Matching Conceptual Graphs as an Aid to Requirements Re-use". *Pages 112–120 of: RE '93: Proceedings of IEEE International Symposium on Requirements Engineering.*

Sabetzadeh, M., & Easterbrook, S. 2003 (October). "Analysis of Inconsistency in Graph-Based Viewpoints: A Category-Theoretic Approach". *Pages 12–21 of: ASE '03: Proceedings of the 18th IEEE International Conference on Automated Software Engineering.*

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

Sabetzadeh, M., Nejati, S., Easterbrook, S., & Chechik, M. 2007a. "A Relationship-Driven Framework for Model Merging". *In: ICSE Workshop on Modeling in Software Engineering (MiSE '07).*

Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., & Chechik, M. 2007b. "Consis-

tency Checking of Conceptual Models via Model Merging". *Pages 221–230 of: RE '07: Proceedings of 15th IEEE International Requirements Engineering Conference.*

Sabetzadeh, M., Nejati, S., Easterbrook, S., & Chechik, M. 2008. "Global Consistency Checking of Distributed Models with TReMer+". *Pages 815–818 of: ICSE '08: Proceedings of the 30th International Conference on Software Engineering.*

Schmidt, D. 2004. "Closed and Logical Relations for Over- and Under-Approximation of Powersets". *Pages 22–37 of: SAS '04: Proceedings of the 11th International Symposium on Static Analysis.* Lecture Notes in Computer Science, vol. 4134. Springer.

Selic, B. 2006. "Model-Driven Development: Its Essence and Opportunities". *Pages 313–319 of: ISORC '06: Proceedings of 9th International Symposium on Object and Component-Oriented Real-Time Distributed Computing.*

Sharygina, N., Browne, J., & Kurshan, R. 2001. "A Formal Object-Oriented Analysis for Software Reliability: Design for Verification". *Pages 318–332 of: FASE '01: Proceedings of 4th Internationsl Conference on Formal Aspects of Software Engineering.*

Shaw, M., & Garlan, D. 1996. *"Software Architecture: Perspectives on an Emerging Discipline".* Prentice-Hall, Inc.

Shoham, S., & Grumberg, O. 2004. "Monotonic Abstraction-Refinement for CTL". *Pages 546–560 of: TACAS '04: Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Lecture Notes in Computer Science, vol. 2988.

Shoham, S., & Grumberg, O. 2006. "3-Valued Abstraction: More Precision at Less Cost". *Pages 399–410 of: LICS '06: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science.*

Sibay, G., Uchitel, S., & Braberman, V. 2008. "Existential Live Sequence Charts Revisited". *Pages 41–50 of: ICSE '08: Proceedings of the 30th International Conference on Software Engineering.*

Sokolsky, O., Kannan, S., & Lee, I. 2006. "Simulation-Based Graph Similarity". *Pages 426–440 of: TACAS '06: Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Lecture Notes in Computer Science, vol. 3920. Springer.

Spanoudakis, G., & Finkelstein, A. 1997. "Reconciling Requirements: A Method for Managing Interference, Inconsistency and Conflict". *Annals of Software Engineering*, **3**, 433–457.

Spivey, J. M. 1989. *"The Z Notation – A Reference Manual"*. Addison.

Stirling, C. 1999. "Bisimulation, Modal Logic and Model Checking Games". *Logic Journal of the IGPL*, **7**(1), 103–124.

Tarr, P., Ossher, H., Harrison, W., & Jr., S. Sutton. 1999. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". *Pages 107–119 of: ICSE '99: Proceedings of the 21st International Conference on Software Engineering.*

Uchitel, S., & Chechik, M. 2004. "Merging Partial Behavioural Models". *Pages 43–52 of: SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering.*

Uchitel, S., Brunet, G., & Chechik, M. 2007. "Behaviour Model Synthesis From Properties and Scenarios". *Pages 34–43 of: ICSE '07: Proceedings of the 29th International Conference on Software Engineering.*

van Breugel, F. 2008. *"ABE '08: Workshop on Approximate Behavioural Equivalences"*.

van Glabbeek, R. 1993. "The Linear Time - Branching Time Spectrum II". *Pages 66–81 of: CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory.*

Wei, O., Gurfinkel, A., & Chechik, M. 2008. *"Mixed Transition Systems Revisited"*. Submitted.

Whittle, J., & Schumann, J. 2000. "Generating Statechart Designs from Scenarios". *Pages 314–323 of: ICSE '00: Proceedings of 22nd International Conference on Software Engineering.* ACM Press.

Whittle, J., Moreira, A., Araújo, J., Jayaraman, P., Elkhodary, A., & Rabbi, R. 2007. "An Expressive Aspect Composition Language for UML State Diagrams". *Pages 514–528 of: MoDELS '07: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems.*

Zave, P. 1999. *"FAQ Sheet on Feature Interaction"*. `http://www.research.att.com/~pamela/faq.html`.

Zave, P., & Jackson, M. 2002. "A Call Abstraction for Component Coordination". *Electronic Notes in Theoretical Computer Science*, **66**(4).

Zimmer, A. 2007. *"Prioritizing Features Through Categorization: An Approach to Resolving Feature Interactions"*. Ph.D. thesis, University of Waterloo.

Zito, A., Diskin, Z., & Dingel, J. 2006. "Package Merge in UML 2: Practice vs. Theory?". *Pages 185–199 of: MoDELS '06: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems.*

Appendix A

Models for the Evaluation in Chapter 4

In this appendix, we first briefly describe the ECharts language – the Statecharts dialect in which the models of Chapter 4 are described. Then, we provide the source models for the case study of Chapter 4, and their final merges.

A.1 The ECharts Language

ECharts (Bond, 2006) is a state machine language that has been developed in AT&T to be used in the design and implementation of telecommunication features. The syntax of ECharts is the same as that of Statecharts – it supports hierarchical state machines, concurrent (orthogonal) state machines, machine synchronization, and fork/join transitions. The semantics of the language is defined so as to enable efficient and automated code generation from ECharts models. In particular, the following decisions are made to determinize and/or add new control over the behaviour of ECharts models: (1) transitions are prioritized; (2) concurrency is defined as interleaving of actions; and (3) intra-communication is only allowed between a state and its sub-states. This form of communication is implemented based on the communication mechanism between proce-

dures via parameter passing. In Section 4.3, we provided a formalization for ECharts that takes into account these three semantic features. More details about the ECharts language is available in (Bond, 2006).

A.2 Statecharts Models

In this section, we provide the source models for the case-study in Section 4.8, and their final merges. These models were originally implemented in the ECharts language. These implementations could be compiled and translated into Java programs using the library `echarts.jar` (see (Bond, 2006)). However, they were not abstract enough for understanding and comparing the models' behaviours. Therefore, we first manually generated graphical Statecharts models from the ECharts implementations and studied the graphical models to sufficiently understand their function. To apply our matcher to the models, we encoded the graphical models in XML. These XML descriptions are available at `http://www.cs.toronto.edu/~shiva/MatchTool/models/`, and have been used as input to our matcher `http://www.cs.toronto.edu/~shiva/MatchTool/`. We then specified the models in TReMer `http://www.cs.toronto.edu/~mehrdad/tremer/` to compute their merges. Below, we provide XML encoding of these models as well as their visual representations in TReMer.

XML descriptions for the Statecharts models used in the case-study in Section 4.8

To represent states and transitions, we use the following XML tags in our encoding: `state` and `transition`. For each state tag, we specify the following attributes: (1) state's id, (2) a flag indicating if the state is an initial state, (3) state's name, (4) state's type (a state's type can be `Atomic` or `superstate`. In the latter case, the state's type is the name of a standard telecom state machine that we have decided not to expand

further), (5) the id of the state's parent state (if the state is root the id is -1), and (6) the depth of the state in its corresponding state hierarchy tree. For each transition tag, we specify the following attributes: (1) the id of the source state, (2) the id of the target state, (3) transition's event, (4) transition's action, and (5) transition's condition. Here, we provide the XML descriptions for the call logger examples.

Call Logger Call Logger (VPLUS version)]

```
-<statemachine id="CallLogger_VPLUS">
<state id="0" initial="true" name="CallLoggerVPLUS" type="CallLoggerVPLUS" parentId="-1" depth="0"/>
<state id="1" initial="true" name="MONITOR_OUTCOME" type="MonitorOutcome" parentId="0" depth="1"/>
<state id="2" initial="false" name="TIMER_STARTED" type=" Transparent2Links" parentId="0" depth="1"/>
<state id="3" initial="false" name="LOG_SUCCESS" type="Atomic" parentId="0" depth="1"/>
<state id="4" initial="false" name="LOG_FAILURE" type="Atomic" parentId="0" depth="1"/>
<state id="5" initial="true" name="INIT2LINKS" type="InitializeTransparent2Links" parentId="1" depth="2"/>
<state id="6" initial="false" name="WAITING_FOR_OUTCOME" type="Transparent2Links" parentId="1" depth="2"/>
<state id="7" initial="false" name="SUCCESS" type="Atomic" parentId="1" depth="2"/>
<state id="8" initial="false" name="FAILURE" type="Atomic" parentId="1" depth="2"/>
<state id="9" initial="true" name="Start" type="Atomic" parentId="5" depth="3"/>
<state id="10" initial="true" name="Source" type="Atomic" parentId="5" depth="3"/>
<state id="11" initial="true" name="Target" type="Atomic" parentId="5" depth="3"/>
<state id="12" initial="false" name="LINK_CALLER" type="Atomic" parentId="5" depth="3"/>
<state id="13" initial="false" name="TRANSPARENT" type="Transparent" parentId="5" depth="3"/>
<state id="14" initial="false" name="OPEN_LINK" type="Open2Links" parentId="5" depth="3"/>
<state id="15" initial="false" name="LINK_OPENED" type="Transparent2Links" parentId="5" depth="3"/>
<state id="16" initial="false" name="LINK_UNOPENED" type="Atomic" parentId="5" depth="3"/>
<state id="17" initial="true" name="TRANSPARENT" type="Transparent" parentId="6" depth="3"/>
<state id="18" initial="false" name="OPEN_LINK" type="Open2Links" parentId="6" depth="3"/>
<transition from="9" to="12" event="setup" condition="zone=source" action="callee=par;caller=sub"/>
<transition from="9" to="12" event="setup" condition="zone=target" action="callee=sub;caller=par"/>
<transition from="10" to="12" event="" condition="" action="callee=par;caller=sub"/>
<transition from="11" to="12" event="" condition="" action="callee=sub;caller=par"/>
<transition from="12" to="13" event="callee?upack" condition="" action=""/>
<transition from="13" to="14" event="caller?open" condition="" action=""/>
<transition from="13" to="16" event="failure" condition="" action=""/>
<transition from="14" to="13" event="Link_Unopened" condition="" action=""/>
<transition from="14" to="15" event="Link_Opened" condition="" action=""/>
<transition from="15" to="6" event="" condition="" action=""/>
<transition from="17" to="18" event="sub?Open" condition="" action="opener=sub;openee=par"/>
```

```
<transition from="17" to="18" event="par?Open" condition="" action="opener=par;openee=sub"/>
<transition from="18" to="18" event="openee?Open" condition="" action="openee?status;opener?status"/>
<transition from="18" to="17" event="Link_Opened" condition="" action=""/>
<transition from="18" to="17" event="Link_Unopened" condition="" action=""/>
<transition from="6" to="8" event="par?Reject" condition="" action=""/>
<transition from="6" to="8" event="sub?Reject" condition="" action=""/>
<transition from="6" to="8" event="par?TearDown" condition="" action=""/>
<transition from="6" to="8" event="sub?TearDown" condition="" action=""/>
<transition from="6" to="7" event="sub?Accept" condition="" action=""/>
<transition from="6" to="7" event="par?Accept" condition="" action=""/>
<transition from="7" to="2" event="" condition="" action=""/>
<transition from="8" to="4" event="V-link" condition="" action=""/>
<transition from="2" to="3" event="sub?TearDown" condition="" action=""/>
<transition from="2" to="3" event="par?TearDown" condition="" action=""/>
        <transition from="16" to="17" event="" condition="" action=""/>
<transition from="17" to="8" event="par?Reject" condition="" action=""/>
<transition from="17" to="8" event="sub?Reject" condition="" action=""/>
<transition from="17" to="8" event="par?TearDown" condition="" action=""/>
<transition from="17" to="8" event="sub?TearDown" condition="" action=""/>
<transition from="17" to="7" event="sub?Accept" condition="" action=""/>
<transition from="17" to="7" event="par?Accept" condition="" action=""/>
<transition from="18" to="8" event="par?Reject" condition="" action=""/>
<transition from="18" to="8" event="sub?Reject" condition="" action=""/>
<transition from="18" to="8" event="par?TearDown" condition="" action=""/>
<transition from="18" to="8" event="sub?TearDown" condition="" action=""/>
<transition from="18" to="7" event="sub?Accept" condition="" action=""/>
<transition from="18" to="7" event="par?Accept" condition="" action=""/>
<transition from="0" to="1" event="contain" condition="" action=""/>
<transition from="0" to="2" event="contain" condition="" action=""/>
<transition from="0" to="3" event="contain" condition="" action=""/>
<transition from="0" to="4" event="contain" condition="" action=""/>
<transition from="1" to="5" event="contain" condition="" action=""/>
<transition from="1" to="6" event="contain" condition="" action=""/>
<transition from="1" to="8" event="contain" condition="" action=""/>
<transition from="1" to="7" event="contain" condition="" action=""/>
<transition from="5" to="9" event="contain" condition="" action=""/>
<transition from="5" to="10" event="contain" condition="" action=""/>
</statemachine>
```

Call Logger (VOIP version)]

```
-<statemachine id="CallLogger">
<state id="0" initial="true" name="CallLoggerACSVOIP" type="CallLoggerACSVOIP" parentId="-2" depth="0"/>
<state id="1" initial="true" name="MONITOR_OUTCOME" type="MonitorOutcome" parentId="0" depth="1"/>
<state id="2" initial="false" name="TIMER_STARTED" type=" Transparent2Links" parentId="0" depth="1"/>
<state id="3" initial="false" name="LOG_SUCCESS" type="Atomic" parentId="0" depth="1"/>
<state id="4" initial="false" name="LOG_FAILURE" type="Atomic" parentId="0" depth="1"/>
<state id="5" initial="false" name="LOG_VOICEMAIL" type="Atomic" parentId="0" depth="1"/>
<state id="6" initial="true" name="Start" type="Atomic" parentId="1" depth="2"/>
<state id="7" initial="false" name="SOURCE_SETUP" type="Atomic" parentId="1" depth="2"/>
<state id="8" initial="false" name="LINK_SUBSCRIBER" type="Atomic" parentId="1" depth="2"/>
<state id="9" initial="false" name="LINK_PARTICIPANT" type="Atomic" parentId="1" depth="2"/>
<state id="10" initial="false" name="WAITING_FOR_OUTCOME" type="Concurrent" parentId="1" depth="2"/>
<state id="11" initial="false" name="SUCCESS" type="Atomic" parentId="1" depth="2"/>
<state id="12" initial="false" name="FAILURE" type="Atomic" parentId="1" depth="2"/>
<state id="13" initial="false" name="VOICEMAIL" type="Atomic" parentId="1" depth="2"/>
<state id="14" initial="true" name="TRANSPARENT" type="Transparent2Links" parentId="10" parallelId="0" depth="3"/>
<state id="15" initial="true" name="STATUS" type="Complex" parentId="10" parallelId="0" depth="3"/>
<state id="16" initial="true" name="WAIT" type="Atomic" parentId="15" depth="4"/>
<state id="17" initial="false" name="SUCCESS" type="Atomic" parentId="15" depth="4"/>
<state id="18" initial="false" name="FAILURE" type="Atomic" parentId="15" depth="4"/>
<state id="19" initial="false" name="VOICEMAIL" type="Atomic" parentId="15" depth="4"/>
<state id="20" initial="true" name="TRANSPARENT" type="Transparent" parentId="14" depth="4"/>
<state id="21" initial="false" name="OPEN_LINK" type="Open2Links" parentId="14" depth="4"/>
<transition from="6" to="7" event="setup" condition="zone=source" action=""/>
<transition from="6" to="8" event="setup" condition="zone=target" action=""/>
<transition from="7" to="9" event="" condition="" action=""/>
<transition from="8" to="10" event="sub?Upack" condition="" action=""/>
<transition from="9" to="10" event="par?Upack" condition="" action=""/>
<transition from="20" to="21" event="sub?Open" condition="" action="Opener=sub;Openee=par"/>
<transition from="20" to="21" event="par?Open" condition="" action="Opener=par;Openee=sub"/>
<transition from="21" to="21" event="Openee?Open" condition="" action=""/>
<transition from="21" to="21" event="Openee?status" condition="" action=""/>
<transition from="21" to="21" event="Opener?status" condition="" action=""/>
<transition from="21" to="20" event="Link_Opened" condition="" action=""/>
<transition from="21" to="20" event="Link_Unopened" condition="" action=""/>
<transition from="16" to="17" event="par?Avail" condition="zone=source" action=""/>
<transition from="16" to="17" event="sub?Avail" condition="zone=target" action=""/>
<transition from="16" to="18" event="par?Unavail" condition="zone=source" action=""/>
<transition from="16" to="18" event="sub?Unavail" condition="zone=target" action=""/>
<transition from="16" to="18" event="par?Reject" condition="zone=source" action=""/>
<transition from="16" to="18" event="sub?Reject" condition="zone=target" action=""/>
```

```
<transition from="16" to="19" event="sub?Userstatus" condition="" action=""/>
<transition from="17" to="11" event="" condition="" action=""/>
<transition from="18" to="12" event="" condition="" action=""/>
<transition from="19" to="13" event="" condition="" action=""/>
<transition from="20" to="11" event="" condition="" action=""/>
<transition from="20" to="12" event="" condition="" action=""/>
<transition from="20" to="13" event="" condition="" action=""/>
<transition from="10" to="12" event="sub?TearDown" condition="" action=""/>
<transition from="10" to="12" event="par?TearDown" condition="" action=""/>
<transition from="11" to="2" event="" condition="" action=""/>
<transition from="12" to="4" event="" condition="" action=""/>
<transition from="13" to="5" event="" condition="" action=""/>
<transition from="2" to="3" event="sub?TearDown" condition="" action=""/>
<transition from="2" to="3" event="par?TearDown" condition="" action=""/>
<transition from="8" to="20" event="sub?Upack" condition="" action=""/>
<transition from="8" to="16" event="sub?Upack" condition="" action=""/>
<transition from="9" to="20" event="par?Upack" condition="" action=""/>
<transition from="9" to="16" event="par?Upack" condition="" action=""/>
<transition from="20" to="11" event="sub?TearDown" condition="" action=""/>
<transition from="20" to="11" event="par?TearDown" condition="" action=""/>
<transition from="21" to="11" event="sub?TearDown" condition="" action=""/>
<transition from="21" to="11" event="par?TearDown" condition="" action=""/>
<transition from="16" to="11" event="sub?TearDown" condition="" action=""/>
<transition from="16" to="11" event="par?TearDown" condition="" action=""/>
<transition from="17" to="11" event="sub?TearDown" condition="" action=""/>
<transition from="17" to="11" event="par?TearDown" condition="" action=""/>
<transition from="18" to="11" event="sub?TearDown" condition="" action=""/>
<transition from="18" to="11" event="par?TearDown" condition="" action=""/>
<transition from="19" to="11" event="sub?TearDown" condition="" action=""/>
<transition from="19" to="11" event="par?TearDown" condition="" action=""/>
<transition from="20" to="12" event="sub?TearDown" condition="" action=""/>
<transition from="20" to="12" event="par?TearDown" condition="" action=""/>
<transition from="21" to="12" event="sub?TearDown" condition="" action=""/>
<transition from="21" to="12" event="par?TearDown" condition="" action=""/>
<transition from="16" to="12" event="sub?TearDown" condition="" action=""/>
<transition from="16" to="12" event="par?TearDown" condition="" action=""/>
<transition from="17" to="12" event="sub?TearDown" condition="" action=""/>
<transition from="17" to="12" event="par?TearDown" condition="" action=""/>
<transition from="18" to="12" event="sub?TearDown" condition="" action=""/>
<transition from="18" to="12" event="par?TearDown" condition="" action=""/>
<transition from="19" to="12" event="sub?TearDown" condition="" action=""/>
```

```
<transition from="19" to="12" event="par?TearDown" condition="" action=""/>
<transition from="20" to="13" event="sub?TearDown" condition="" action=""/>
<transition from="20" to="13" event="par?TearDown" condition="" action=""/>
<transition from="21" to="13" event="sub?TearDown" condition="" action=""/>
<transition from="21" to="13" event="par?TearDown" condition="" action=""/>
<transition from="16" to="13" event="sub?TearDown" condition="" action=""/>
<transition from="16" to="13" event="par?TearDown" condition="" action=""/>
<transition from="0" to="1" event="contain" condition="" action=""/>
<transition from="0" to="2" event="contain" condition="" action=""/>
<transition from="0" to="3" event="contain" condition="" action=""/>
<transition from="0" to="4" event="contain" condition="" action=""/>
<transition from="1" to="6" event="contain" condition="" action=""/>
<transition from="1" to="7" event="contain" condition="" action=""/>
<transition from="1" to="8" event="contain" condition="" action=""/>
<transition from="1" to="9" event="contain" condition="" action=""/>
<transition from="1" to="10" event="contain" condition="" action=""/>
<transition from="1" to="11" event="contain" condition="" action=""/>
<transition from="1" to="12" event="contain" condition="" action=""/>
</statemachine>
```

Relationship between Call Logger VPLUS and Call Logger VOIP versions

(0,0),

(1,1),

(9,6),

(12,7),

(12,8),

(12,9),

(7,11),

(8,12),

(2,2),

(3,3),

(4,4)

Visual representation of the Statecharts models used in the case-study in Section 4.8

Figures A.1 to A.3 show the Call Logger variants as specified in TReMer, and their merge.

Figures A.4 to A.6 show the Parallel Location variants as specified in TReMer, and their merge.

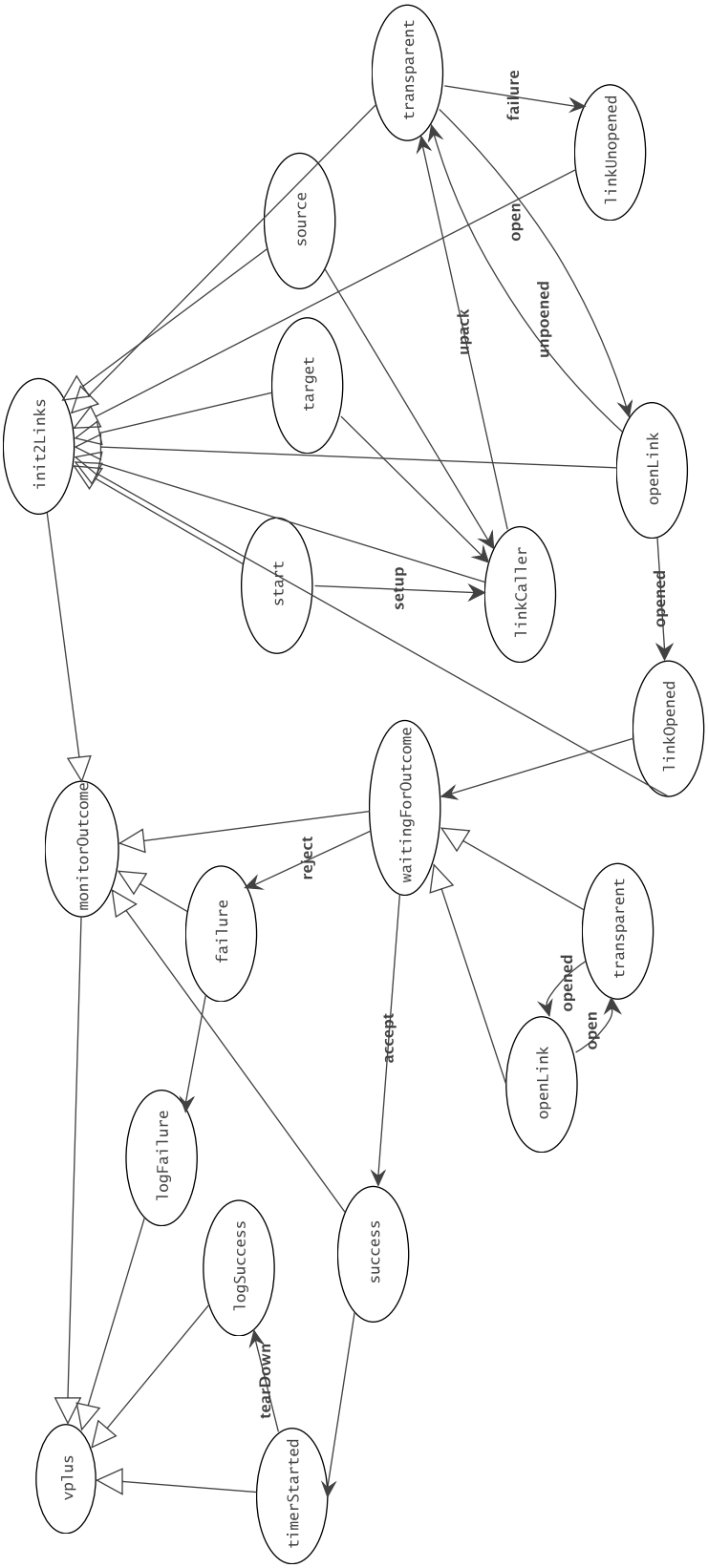


Figure A.1: Call logger vplus version.

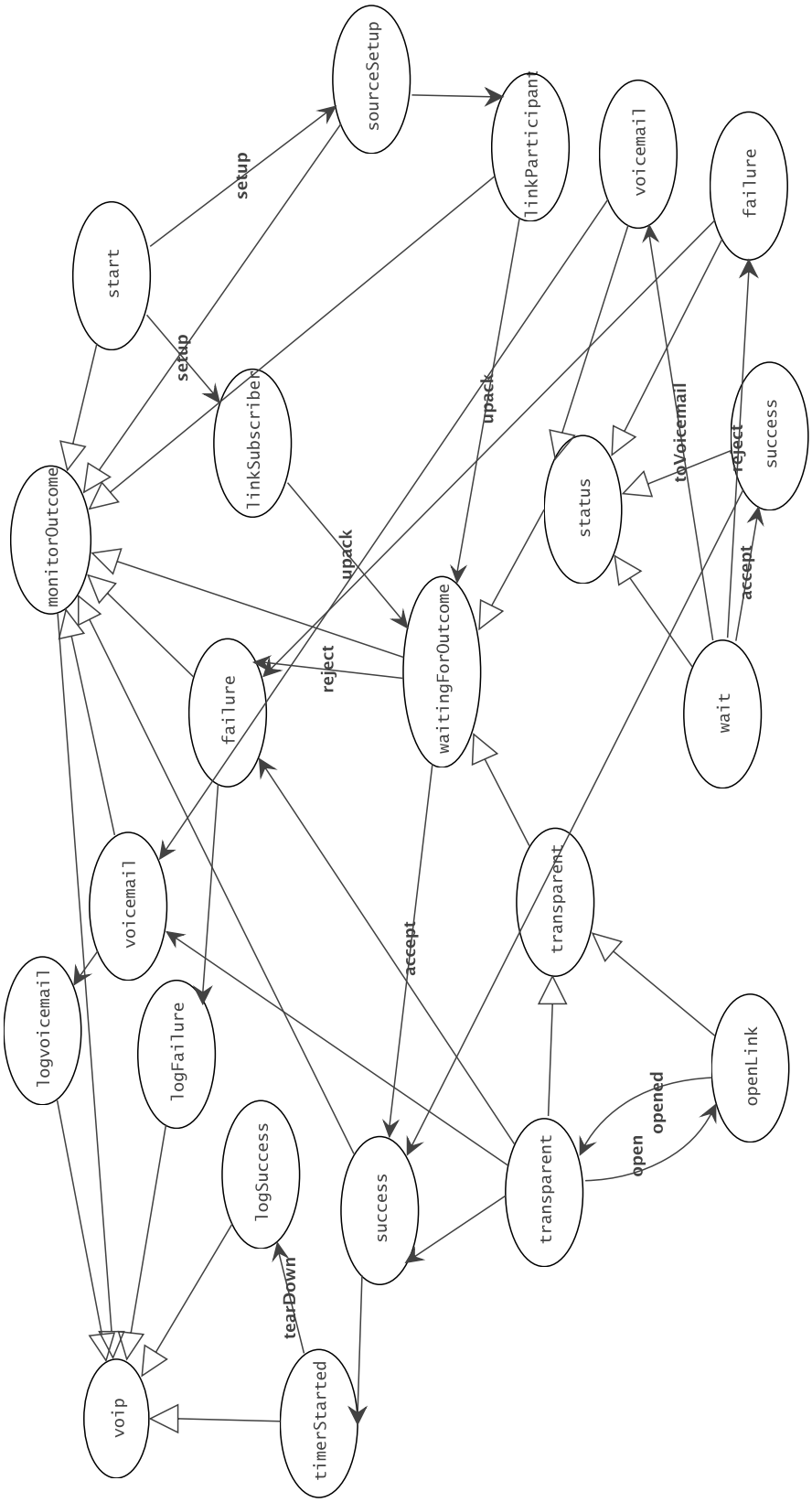


Figure A.2: Call logger voip version.

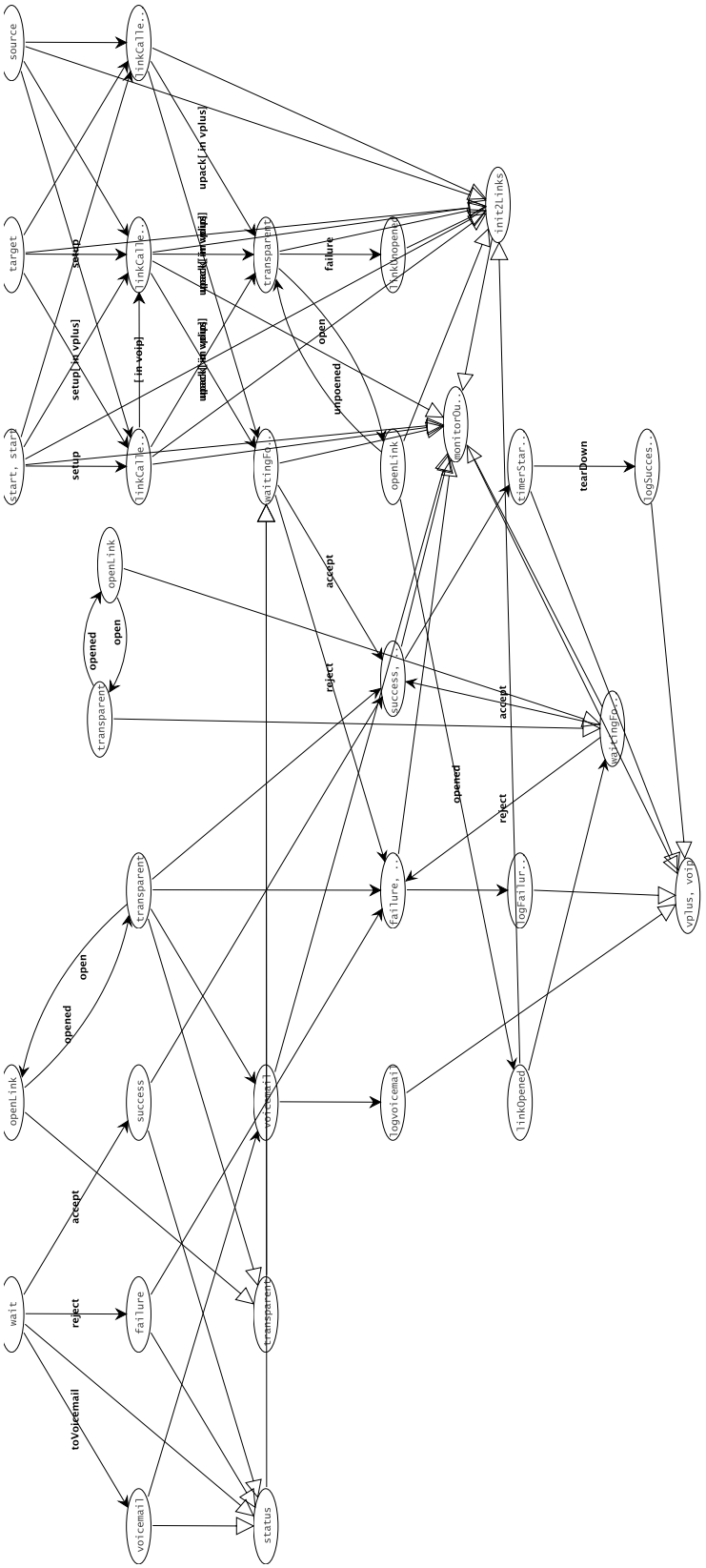


Figure A.3: Merge of vplus and voip for call logger.

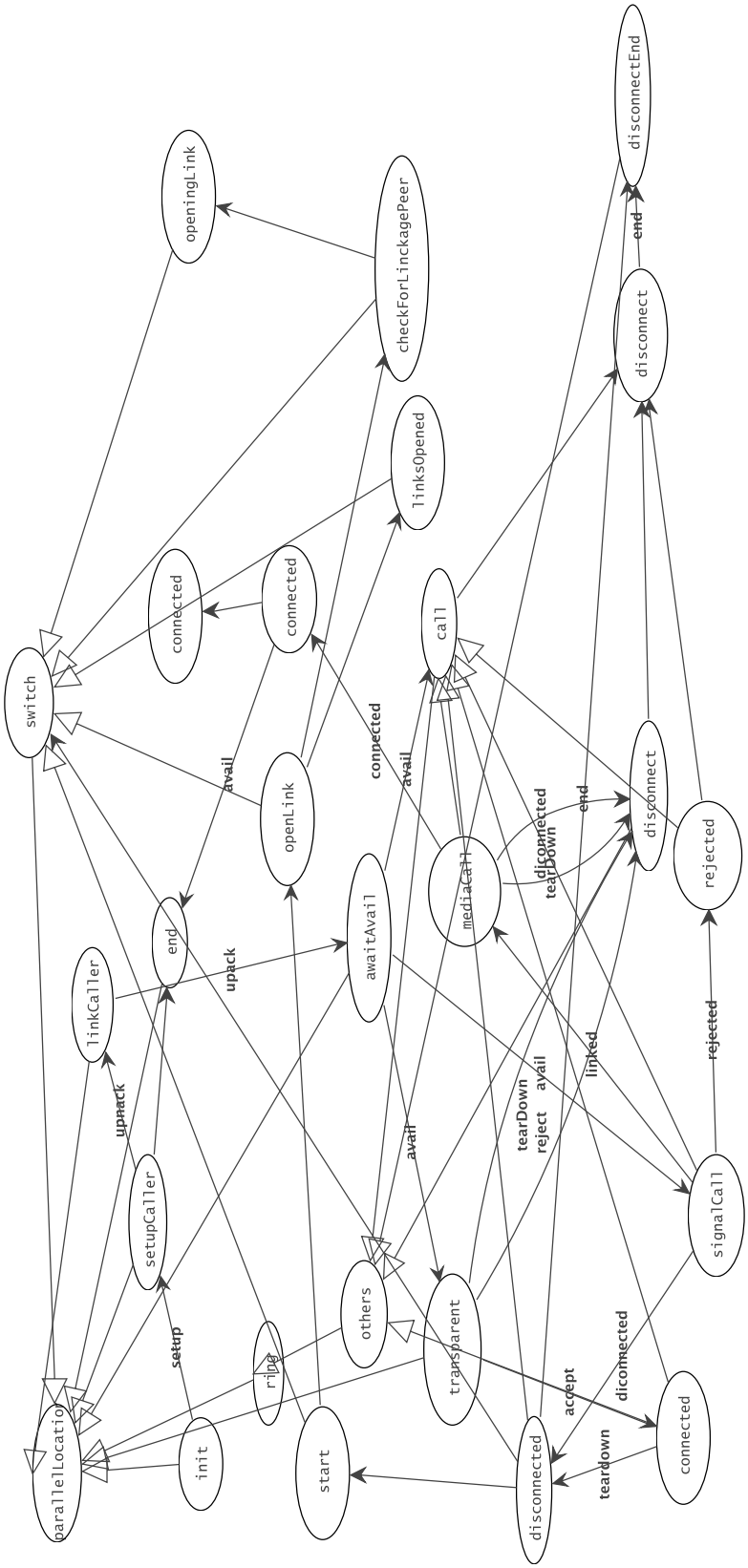


Figure A.4: Parallel location vplus version.

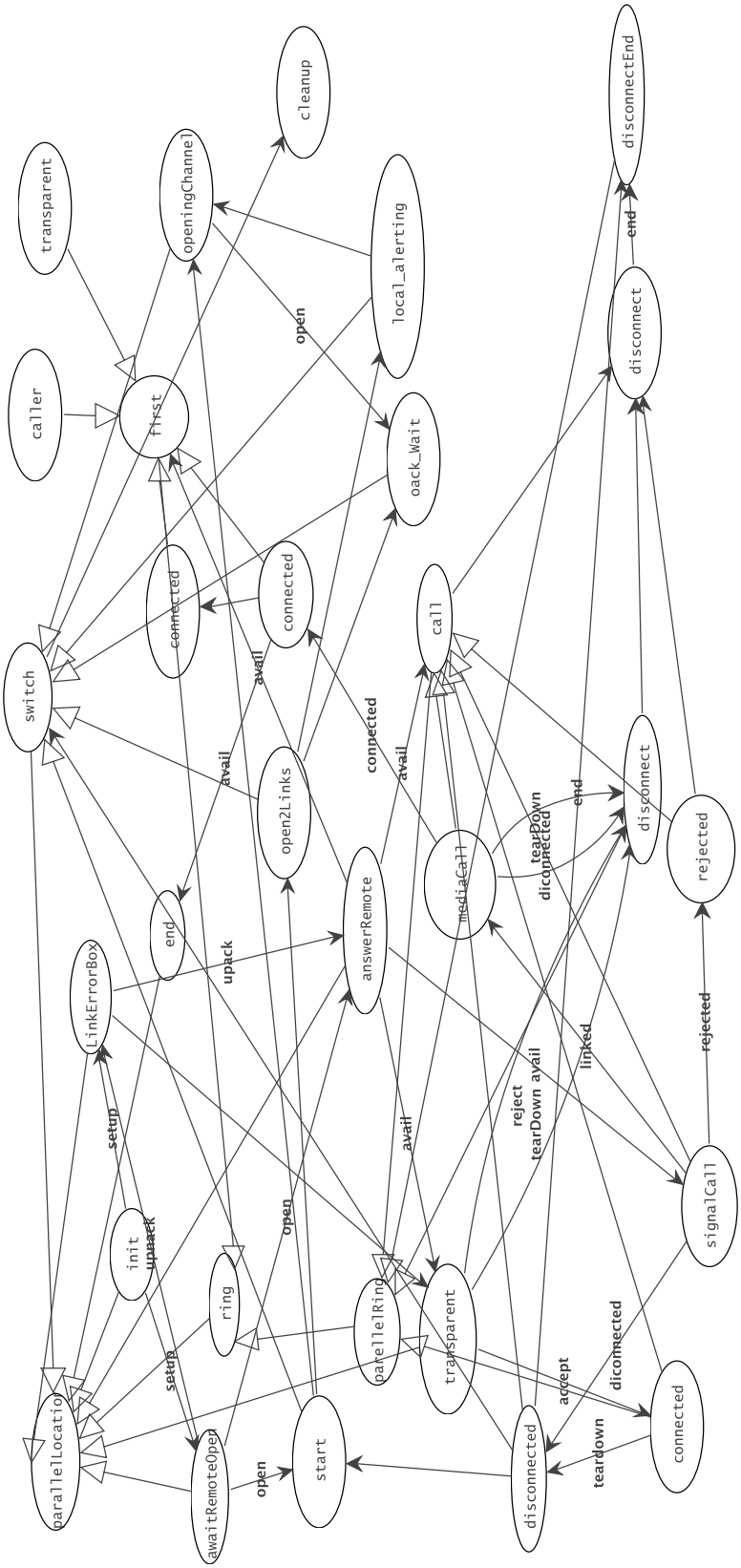


Figure A.5: Parallel location voip version.

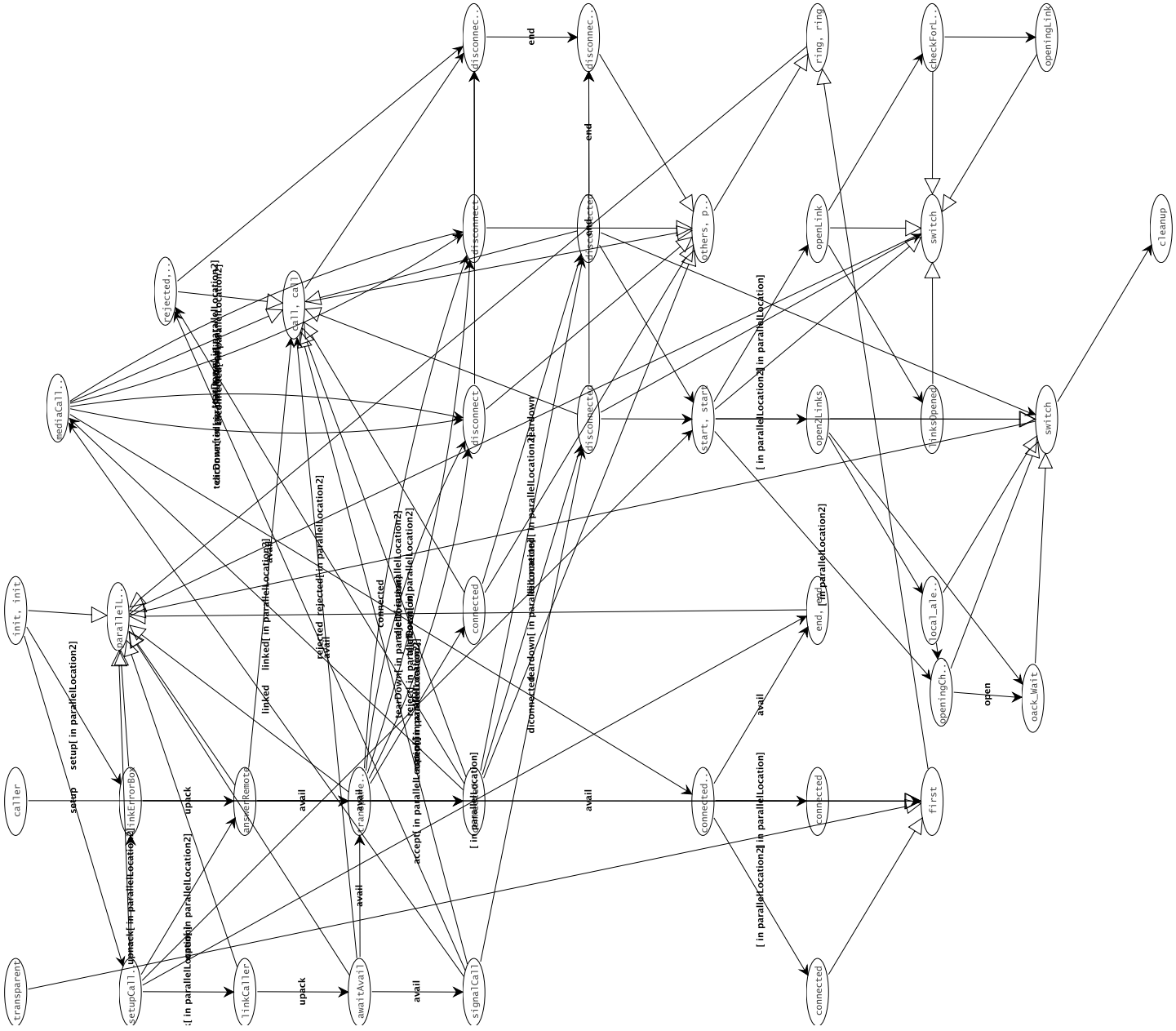


Figure A.6: Merge of vplus and voip for parallel location.

Appendix B

Models for the Evaluation in Chapter 5

In this appendix, we briefly introduce the Boxtalk language, a domain-specific language for specifying telecom features (Zave & Jackson, 2002). Then, we describe the six Boxtalk features studied in Chapter 5 and show that they all implement the transparency pattern. Finally we provide the FSP models used in the evaluation of Chapter 5.

B.1 The Boxtalk Language

Boxtalk is a state machine-based domain-specific language for specifying telecommunication features (Zave & Jackson, 2002). Boxtalk is more abstract than ECharts. The main motivation of Boxtalk is to manage programming complexity with high-level, domain-specific abstractions rather than using nested or parallel states used in ECharts. In Section 5.3, we described how Boxtalk features can be translated to I/O automata. Here, we only note some important technical considerations about the translation.

- States in BoxTalk are explicitly typed as *stable*, *transient*, or *terminal*. While these types do not play a role in feature interaction analysis, they can be distinguished in

I/O automata if necessary: stable states are quiescent I/O automata states that can be exited by some action other than teardown; transient states are non-quiescent I/O automata states; and terminal states are quiescent I/O automata states that cannot be exited by any action other than teardown.

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

  In Boxtalk, a shorthand called *signal-linkage self-loop* is designed to directly connect the right and left ports of a feature, allowing features to pass arbitrary signals form their left neighbour to their right neighbour, and vice versa. Using signal-linkages, Boxtalk models do not need to make their vocabulary explicit. However, in general-purpose formalisms such as I/O automata, we need to determine the vocabulary of models statically and explicitly label their transitions with appropriate actions from their vocabulary. In this thesis, we determined the vocabulary of the I/O automata translations of Boxtalk features for specific instances of telecom usages.

- In DFC, architectural links, i.e., bindings, are dynamic, allowing features to change roles at runtime. Boxtalk represents bindings between features as dynamic *call variables* that can be created, destroyed, and reassigned at runtime. Since I/O automata do not provide any means for describing bindings, we had to extend them

as follows: First, we added a notion of action type to I/O automata to explicitly indicate which action belongs to which call variable. We then implemented bindings between I/O automata using a relabeling mechanism (Magee & Kramer, 2006). However, we still could not capture dynamic aspects of DFC bindings because our typing and relabeling mechanisms are static.

B.2 Boxtalk Models

In this section, we provide the source models for the case-study in Section 5.7. We specify transparent behaviours of these models and provide implementation of these models in LTSA. Figures B.1 to B.2 show the features QT, AC, NATO, and SFM, respectively. The transparent behaviours of these state machines are as follows:

- QT: This feature behaves transparently when it is not enabled, or when it is enabled but the call is urgent. Otherwise, QT tears down the call. (Transparent behaviour: $s_0 \rightarrow s_1 \rightarrow s_3$).
- AC: This feature is always transparent and its service, i.e., confirming if the callee is available, is performed through its internal behaviour. (Transparent behaviour: $s_0 \rightarrow s_1 \rightarrow s_3$ and $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_4$).
- NATO: This feature behaves transparently when the time out message is not triggered. (Transparent behaviour: $s_0 \rightarrow s_1 \rightarrow s_2$).
- SFM: This feature sequentially tries different locations to find the subscriber. This feature behaves transparently when the callee has only one physical phone address. In other words, when SFM only needs to try one location. (Transparent behaviour: $s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_4$ and $s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_5$).

Figures B.3 to B.8 show the implementations of the features CB, RVM, QT, AC, NATO, and SFM in LTSA.

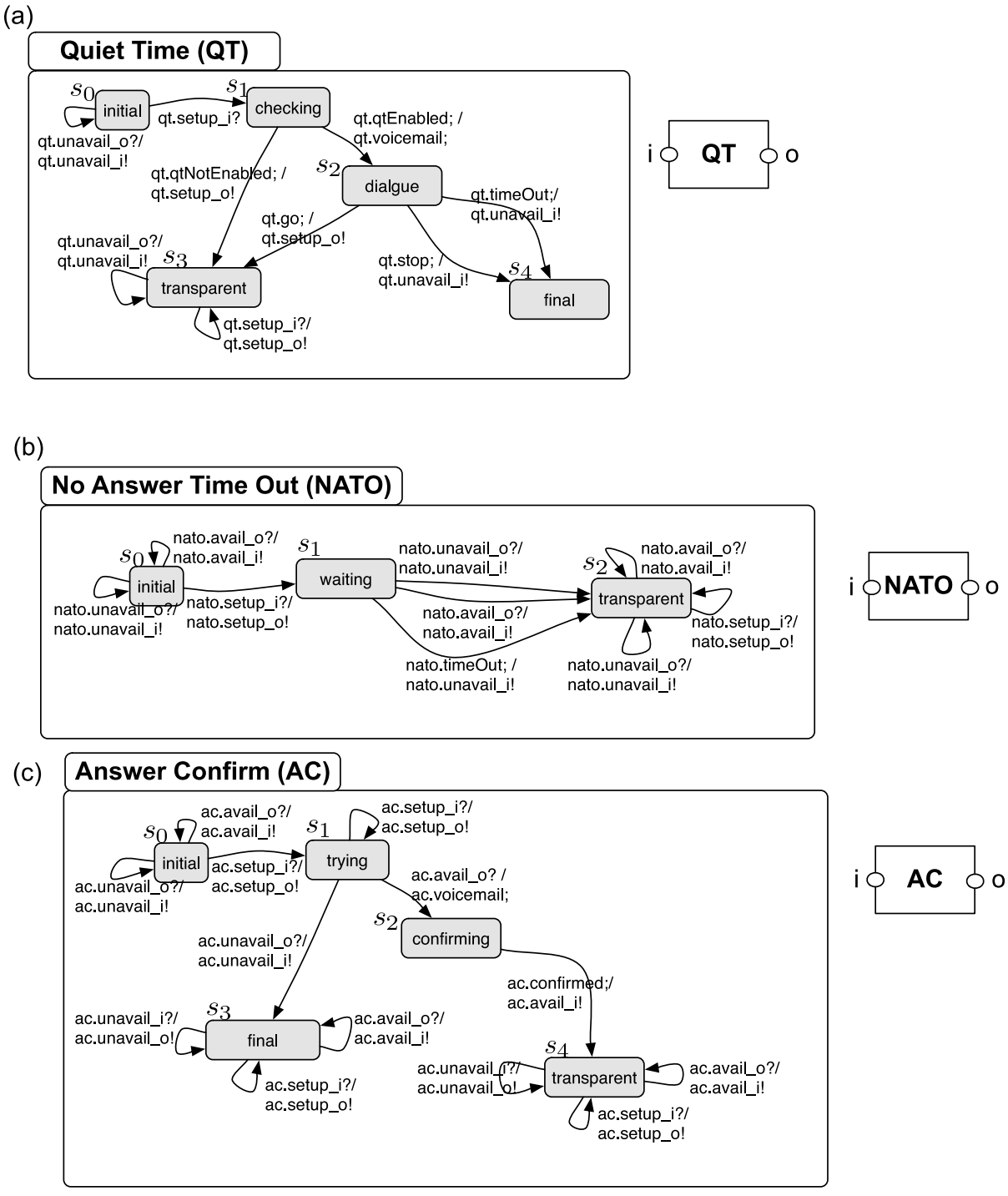


Figure B.1: Boxtalk models: QT, NATO, AC

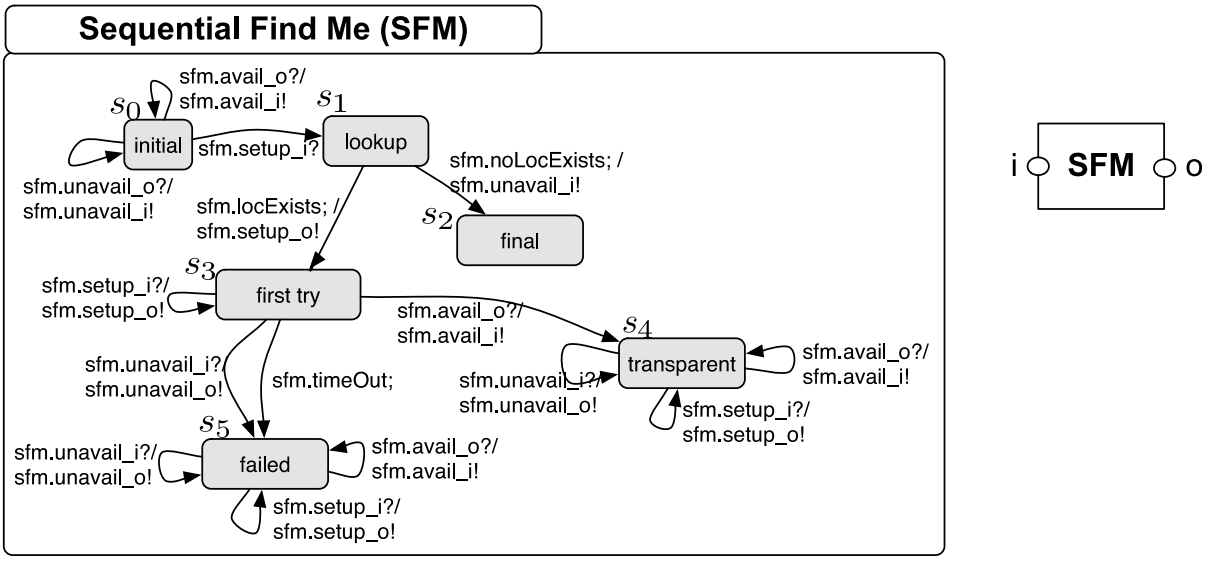


Figure B.2: Boxtalk model: SFM

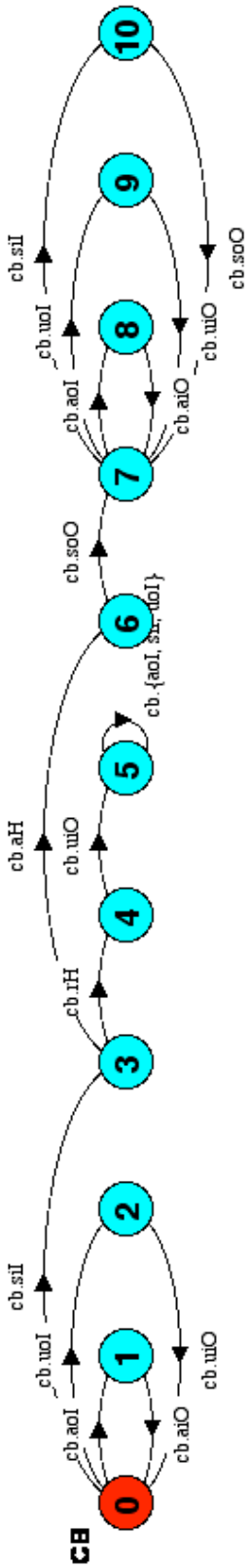


Figure B.3: LTS implemented in LTSA for CB.

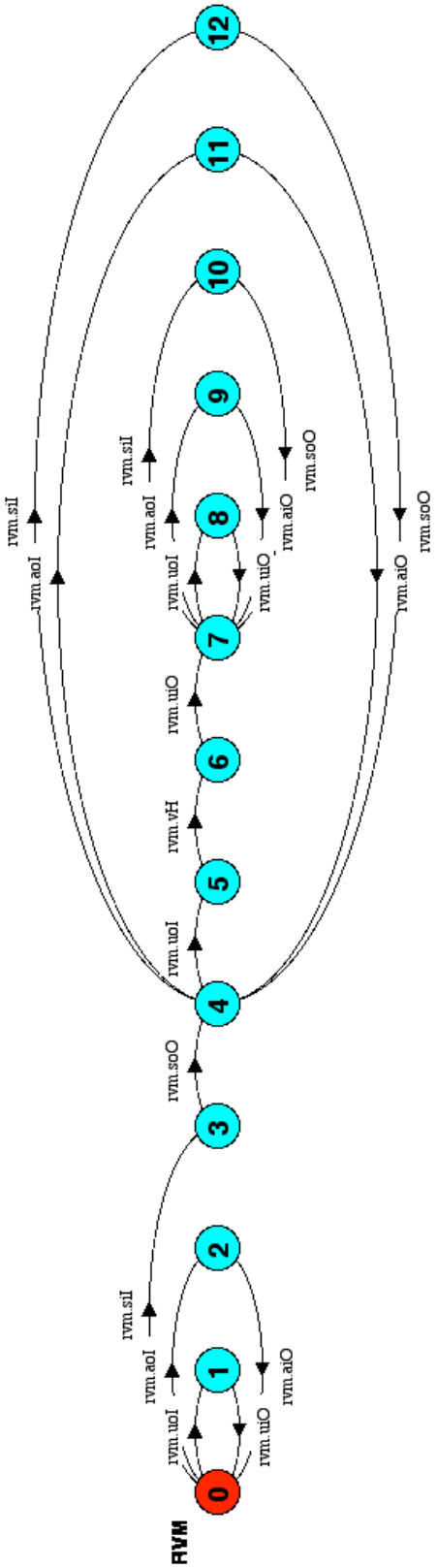


Figure B.4: LTS implemented in LTSA for RVM.

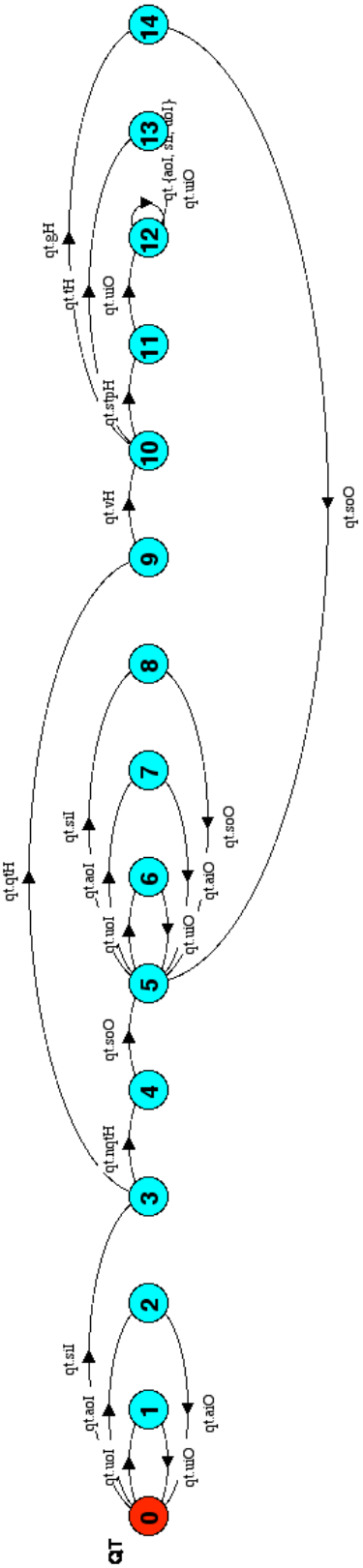


Figure B.5: LTS implemented in LTSA for QT.

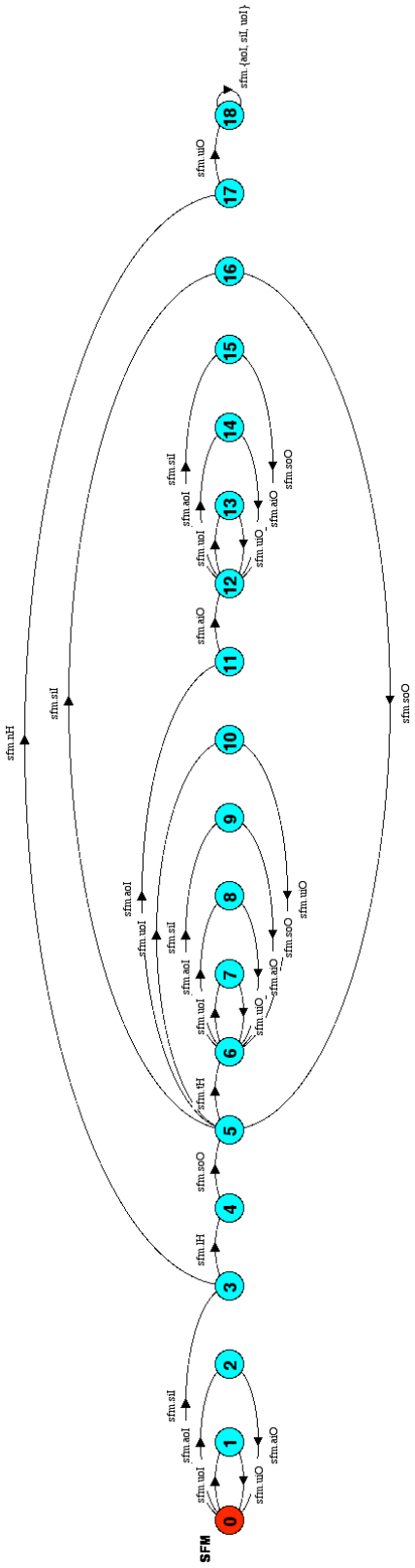


Figure B.6: LTS implemented in LTSA for SFM.

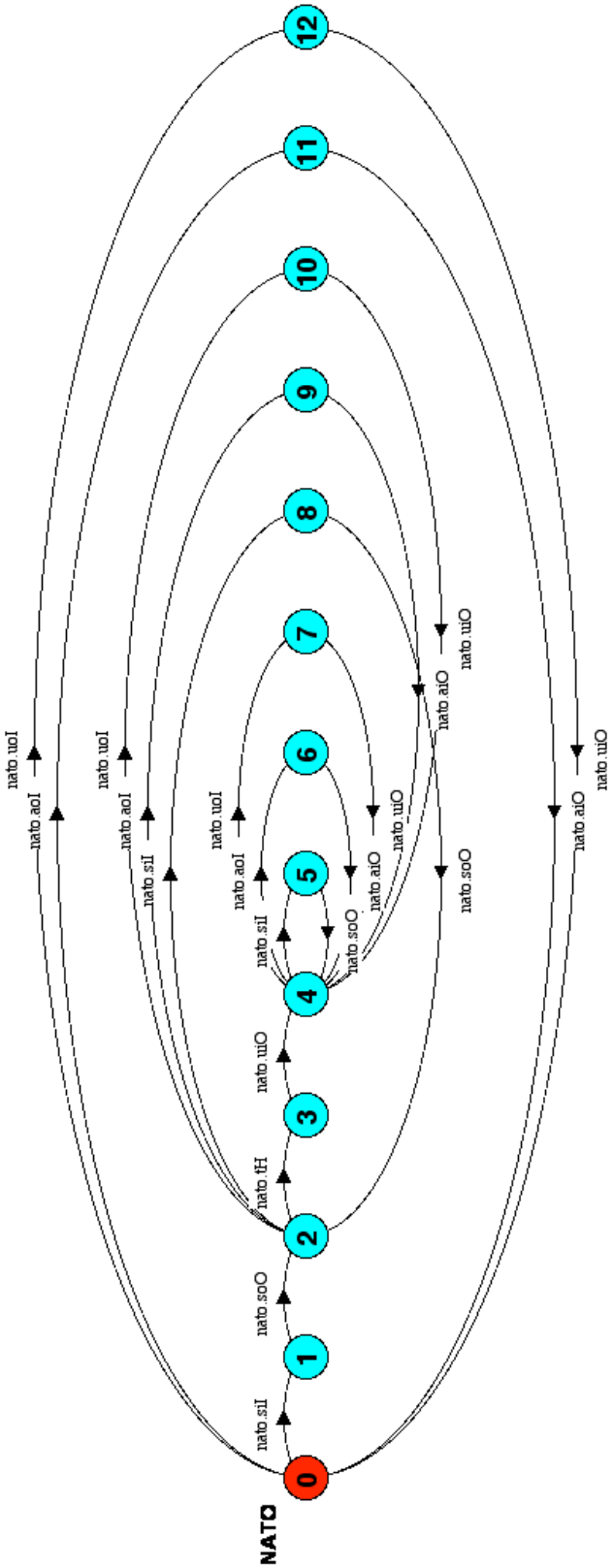


Figure B.7: LTS implemented in LTSA for NATO.

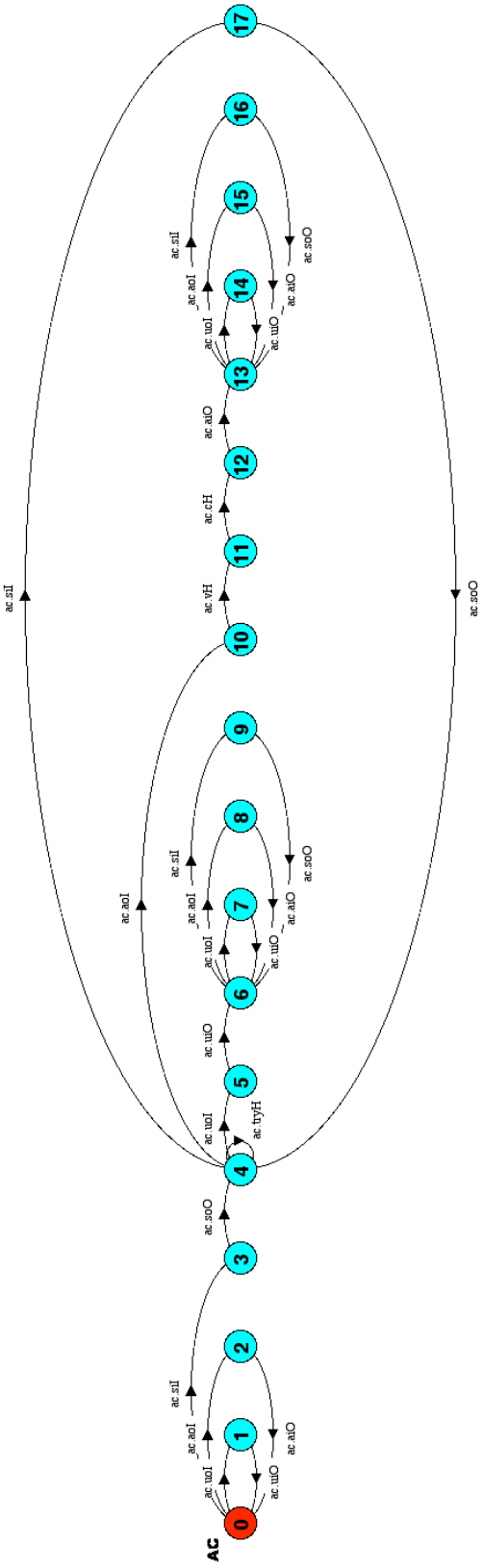


Figure B.8: LTS implemented in LTSA for AC.

Index

3-valued logic, 20

Action-Based Behavioural Model, 18

Behavioural Matching, 75

Behavioural Model, 4

Bisimulation, 31

Boxtalk, 183

CTL, 25

ECharts, 169

I/O Automata, 118

Kripke Modal Transition System (KMTS), 20

Kripke Structure, 19

Labeled Transition System (LTS), 22

Merging KMTSs, 48

Merging Statecharts, 79

Mixed LTSs, 70

Mixed Transition System (MixTS), 23

Model Composition, 2

Model Fusion, 2

Model Merging, 2

Model Weaving, 3

Propositional μ-Calculus (L_μ), 24

R^{may}, 21

R^{must}, 21

Refinement over KMTSs, 32

Refinement over MixTSs, 34

Semantics of KMTS, 25

Semantics of MixTS, 30

Simulation, 33

State-Based Behavioural Model, 18

Statecharts, 65

Trace, 22